JEP 393: Foreign-Memory Access API (Third 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	M
Duration	M
Relates to	JEP 383: Foreign-Memory Access API (Second Incubator) JEP 370: Foreign-Memory Access API (Incubator) JEP 412: Foreign Function & Memory API (Incubator) JEP 389: Foreign Linker API (Incubator)
Reviewed by	Jorn Vernee, Paul Sandoz
Created	2020/09/21 09:58
Updated	2022/03/02 17:09
Issue	8253415

Summary

Introduce an API to allow Java programs to safely and efficiently access foreign memory outside of the Java heap.

History

The Foreign-Memory Access API was first proposed by JEP 370 and targeted to Java 14 in late 2019 as an incubating API, and later re-incubated by JEP 383 which was targeted to Java 15 in mid 2020. This JEP proposes to incorporate refinements based on feedback, and to re-incubate the API in Java 16. The following changes are included in this API refresh:

- A clearer separation of roles between the `MemorySegment` and `MemoryAddress` interfaces;
- A new interface, `MemoryAccess`, which provides common static memory accessors so as to minimize the need for the `VarHandle` API in simple cases;
- Support for *shared* segments; and
- The ability to register segments with a `Cleaner`.

Goals

- *Generality:* A single API should be able to operate on various kinds of foreign memory (e.g., native memory, persistent memory, managed heap memory, etc.).
- *Safety:* It should not be possible for the API to undermine the safety of the JVM, regardless of the kind of memory being operated upon.
- *Control:* Clients should have options as to how memory segments are to be deallocated: either explicitly (via a method call) or implicitly (when the segment is no longer in use).
- *Usability:* For programs that need to access foreign memory, the API should be a compelling alternative to legacy Java APIs such as `sun.misc.Unsafe`.

Non-Goals

- It is not a goal to re-implement legacy Java APIs, such as `sun.misc.Unsafe`, on top of the Foreign-Memory Access API.

Motivation

Many Java programs access foreign memory, such as Ignite, mapDB, memcached, Lucene, and Netty's ByteBuf API. By doing so they can:

- Avoid the cost and unpredictability associated with garbage collection (especially when maintaining large caches);
- Share memory across multiple processes; and
- Serialize and deserialize memory content by mapping files into memory (via, e.g., `mmap`).

Unfortunately, the Java API does not provide a satisfactory solution for accessing foreign memory:

- The `ByteBuffer` API, introduced in Java 1.4, allows the creation of *direct* byte buffers, which are allocated off-heap and therefore allow users to manipulate off-heap memory directly from Java. However, direct buffers are limited. For example, it is not possible to create a direct buffer larger than two gigabytes since the `ByteBuffer` API uses an `int`-based indexing scheme. Moreover, working with direct buffers can be cumbersome since deallocation of the memory associated with them is left to the garbage collector; that is, only after a direct buffer is deemed unreachable by the garbage collector can the associated memory be released. Many requests for enhancement have been filed over the years in order to overcome these and other limitations (e.g., 4496703, 6558368, 4837564 and 5029431). Many of these limitations stem from the fact that the `ByteBuffer` API was designed not only for off-heap memory access but also for producer/consumer exchanges of bulk data in areas such as charset encoding/decoding and partial I/O operations.

- Another common avenue by which developers can access foreign memory from Java code is the `Unsafe` API. `Unsafe` exposes many memory access operations (e.g., `Unsafe::getInt` and `putInt`) which work for on-heap as well as off-heap access thanks to a relatively general addressing model. Using `Unsafe` to access memory is extremely efficient: All memory access operations are defined as HotSpot JVM intrinsics, so memory access operations are routinely optimized by the HotSpot JIT compiler. However, the `Unsafe` API is, by definition, *unsafe* — it allows access to *any* memory location (e.g., `Unsafe::getInt` takes a `long` address). This means a Java program can crash the JVM by accessing some already-freed memory location. On top of that, the `Unsafe` API is not a supported Java API, and its use has always been strongly discouraged.

- Using JNI to access foreign memory is a possibility, but the inherent costs associated with this solution make it seldom applicable in practice. The overall development pipeline is complex, since JNI requires the developer to write and maintain snippets of C code. JNI is also inherently slow, since a Java-to-native transition is required for each access.

In summary, when it comes to accessing foreign memory, developers are faced with a dilemma: Should they choose a safe but limited (and possibly less efficient) path, such as the `ByteBuffer` API, or should they abandon safety guarantees and embrace the dangerous and unsupported `Unsafe` API?

This JEP introduces a safe, supported, and efficient API for foreign memory access. By providing a targeted solution to the problem of accessing foreign memory, developers will be freed of the limitations and dangers of existing APIs. They will also enjoy improved performance, since the new API will be designed from the ground up with JIT optimizations in mind.

Description

The Foreign-Memory Access API is provided as an incubator module named `jdk.incubator.foreign`, in a package of the same name. It introduces three main abstractions: `MemorySegment`, `MemoryAddress`, and `MemoryLayout`:

- A `MemorySegment` models a contiguous memory region with given spatial and temporal bounds,
- A `MemoryAddress` models an address, which could reside either on- or off-heap, and
- A `MemoryLayout` is a programmatic description of a memory segment's contents.

Memory segments can be created from a variety of sources, such as native memory buffers, memory-mapped files, Java arrays, and byte buffers (either direct or heap-based). For instance, a native memory segment can be created as follows:

```
try (MemorySegment segment = MemorySegment.allocateNative(100)) {
   ...
}
```

This will create a memory segment that is associated with a native memory buffer whose size is 100 bytes.

Memory segments are *spatially bounded*, which means they have lower and upper bounds. Any attempt to use the segment to access memory outside of these bounds will result in an exception.

As evidenced by the use of the `try`-with-resource construct above, memory segments are also *temporally bounded*, which means they must be created, used, and then closed when no longer in use. Closing a segment can result in additional side effects, such as deallocation of the memory associated with the segment. Any attempt to access an already-closed memory segment results in an exception. Together, spatial and temporal bounding guarantee the safety of the Foreign-Memory Access API and thus guarantee that its use cannot crash the JVM.

Memory dereference

Dereferencing the memory associated with a segment is achieved by obtaining a *var handle*, an abstraction for data access introduced in Java 9. In particular, a segment is dereferenced with a *memory-access var handle*. This kind of var handle features a pair of *access coordinates*:

- A coordinate of type `MemorySegment` — the segment whose memory is to be dereferenced, and
- A coordinate of type `long` — the offset, from the segment's base address, at which dereference occurs.

Memory-access var handles are obtained using factory methods in the `MemoryHandles` class. For instance, to set the elements of a native memory segment, we could use a memory-access var handle as follows:

```
VarHandle intHandle = MemoryHandles.varHandle(int.class,
        ByteOrder.nativeOrder());

try (MemorySegment segment = MemorySegment.allocateNative(100)) {
    for (int i = 0; i < 25; i++) {
        intHandle.set(segment, i * 4, i);
    }
}
```

More advanced access idioms can be expressed by combining plain memory-access var handles using one or more of the combinator methods provided by the `MemoryHandles` class. With these a client can, for instance, map the memory-access var handle type using a projection/embedding method-handle pair. A client can also reorder the coordinates of a given memory-access var handle, drop one or more coordinates, and insert new coordinates.

To make the API more accessible, the `MemoryAccess` class provides a number of static accessors which can be used to dereference memory segments without the need to construct memory-access var handles. For instance, there is an accessor to set an `int` value in a segment at a given offset, allowing the above example to be simplified as follows:

```
try (MemorySegment segment = MemorySegment.allocateNative(100)) {
    for (int i = 0; i < 25; i++) {
        MemoryAccess.setIntAtOffset(segment, i * 4, i);
    }
}
```

Memory layouts

To enhance the expressiveness of the API, and to reduce the need for explicit numeric computations such as those in the above examples, a `MemoryLayout` can be used to programmatically describe the content of a `MemorySegment`. For instance, the layout of the native memory segment used in the above examples can be described in the following way:

```
SequenceLayout intArrayLayout
    = MemoryLayout.ofSequence(25,
        MemoryLayout.ofValueBits(32,
            ByteOrder.nativeOrder()));
```

This creates a *sequence* memory layout in which a given element layout (a 32-bit value) is repeated 25 times. Once we have a memory layout, we can get rid of all the manual numeric computation in our code and also simplify the creation of the required memory access var handles, as shown in the following example:

```
SequenceLayout intArrayLayout
    = MemoryLayout.ofSequence(25,
        MemoryLayout.ofValueBits(32,
            ByteOrder.nativeOrder()));

VarHandle indexedElementHandle
    = intArrayLayout.varHandle(int.class,
        PathElement.sequenceElement());

try (MemorySegment segment = MemorySegment.allocateNative(intArrayLayout)) {
    for (int i = 0; i < intArrayLayout.elementCount().getAsLong(); i++) {
        indexedElementHandle.set(segment, (long) i, i);
    }
}
```

In this example, the layout object drives the creation of the memory-access var handle through the creation of a *layout path*, which is used to select a nested layout from a complex layout expression. The layout object also drives the allocation of the native memory segment, which is based upon size and alignment information derived from the layout. The loop constant in the previous examples (25) has been replaced with the sequence layout's element count.

Unchecked segments

Dereference operations are only possible on memory segments. Since a memory segment has spatial and temporal bounds, the runtime can always ensure that memory associated with a given segment is dereferenced safely. But there are situations in which clients might only have a memory address; for instance, this is often the case when interacting with native code. Moreover, a memory address can be constructed from `long` values (via the `MemoryAddress::ofLong` factory). In such cases, since the runtime has no way to know the spatial and temporal bounds associated with the memory address, dereferencing memory addresses is forbidden by the API.

To dereference a memory address, a client has two options. If the address is known to fall within a memory segment, the client can perform a *rebase* operation (`MemoryAddress::segmentOffset`). The rebasing operation re-interprets the address's offset relative to the segment's base address to yield a new offset which can be applied to the existing segment — which can then be safely dereferenced.

Alternatively, if no such segment exists then the client can create one unsafely, using the special `MemoryAddress::asSegmentRestricted` factory. This factory effectively attaches spatial and temporal bounds to an otherwise unchecked address, so as to allow dereference operations. As the name suggests, this operation is unsafe by its very nature, and so it must be used with care. For this reason, the Foreign Memory Access API only allows calls to this factory when the JDK system property `foreign.restricted` is set to a value other than deny. The possible values for this property are:

- `deny` — issues a runtime exception on each restricted call (this is the default value);
- `permit` — allows restricted calls;
- `warn` — like `permit`, but also prints a one-line warning on each restricted call; and
- `debug` — like `permit`, but also dumps the stack corresponding to any given restricted call.

We plan, in the future, to integrate access to restricted operations with the module system. Certain modules might somehow declare that they *require* restricted native access. When an application which depends on such modules is executed, the user might need to provide permissions to those modules to perform restricted native operations, or else the runtime will refuse to run the application.

Confinement

In addition to spatial and temporal bounds, segments also feature *thread confinement*. That is, a segment is *owned* by the thread which created it, and no other thread can access the contents of the segment, or perform certain operations (such as `close`) on it. Thread confinement, while restrictive, is crucial to guarantee optimal memory-access performance even in a multi-threaded environment.

The Foreign-Memory Access API provides several ways to relax thread-confinement barriers. First, threads can cooperatively share segments by performing explicit *handoff* operations, in which a thread releases its ownership on a given segment and transfers it to another thread. Consider the following code:

```
MemorySegment segmentA = MemorySegment.allocateNative(10); // confined to thread A
...
var segmentB = segmentA.withOwnerThread(threadB); // now confined to thread B
```

This pattern of access is also known as *serial confinement* and can be useful in producer/consumer use cases where only one thread at a time needs to access a segment. Note that, to make the handoff operation safe, the API *kills* the original segment (as if `close` was called, but without releasing the underlying memory) and returns a new segment with the correct owner. The implementation also makes sure that all writes by the first thread are flushed to memory by the time the second thread accesses the segment.

When serial confinement is not enough then clients can optionally remove thread ownership, that is, turn a confined segment into a *shared* one which can be accessed — and closed — concurrently, by multiple threads. As before, sharing a segment kills the original segment and returns a new segment with no owner thread:

```
MemorySegment segmentA = MemorySegment.allocateNative(10); // confined by thread A
...
var sharedSegment = segmentA.withOwnerThread(null); // now a shared segment
```

A shared segment is especially useful when multiple threads need to operate on the segment's contents in parallel (e.g., using a framework such as Fork/Join) since a `Spliterator` instance can be obtained from a memory segment. For instance, to sum all the 32-bit values of a memory segment in parallel, we can use the following code:

```
SequenceLayout seq = MemoryLayout.ofSequence(1_000_000, MemoryLayouts.JAVA_INT);
SequenceLayout seq_bulk = seq.reshape(-1, 100);
VarHandle intHandle = seq.varHandle(int.class, sequenceElement());

int sum = StreamSupport.stream(MemorySegment.spliterator(segment.withOwnerThread(null),
                                                         seq_bulk),
                               true)
                .mapToInt(slice -> {
                    int res = 0;
                    for (int i = 0; i < 100 ; i++) {
                        res += MemoryAccess.getIntAtIndex(slice, i);
                    }
                    return res;
                }).sum();
```

The `MemorySegment::spliterator` method takes a segment and a *sequence* layout and returns a spliterator instance which splits the segment into chunks which correspond to the elements in the provided sequence layout. Here, we want to sum elements in an array which contains a million elements. Doing a parallel sum where each computation processes exactly one element would be inefficient, so instead we use the layout API to derive a *bulk* sequence layout. The bulk layout is a sequence layout which has the same size as the original layout, but where the elements are arranged into groups of 100 elements — which make it more amenable to parallel processing.

Once we have the spliterator, we can use it to construct a parallel stream and sum the contents of the segment in parallel. Since the segment operated upon by the spliterator is shared, the segment can be accessed from multiple threads concurrently. The spliterator API ensures that access occurs in a regular fashion: It creates slices from the original segment and gives each slice to a thread to perform the desired computation, thus ensuring that no two threads can ever operate concurrently on the same memory region.

Shared segments can also be useful to perform serial confinement in cases where the thread handing off the segment does not know which other thread will continue the work on the segment, for instance:

```
// thread A
MemorySegment segment = MemorySegment.allocateNative(10); // confined by thread A
// do some work
segment = segment.withOwnerThread(null);

// thread B
segment.withOwnerThread(Thread.currentThread()); // now confined by thread B
// do some more work
```

Multiple threads can race to acquire a given shared segment, but the API ensures that only one of them will succeed in acquiring ownership of the shared segment.

Implicit deallocation

Memory segments feature deterministic deallocation but they can also be registered with a `Cleaner` to make sure that the memory resources associated with a segment are released when the garbage collector determines that the segment is no longer reachable:

```
MemorySegment segment = MemorySegment.allocateNative(100);
Cleaner cleaner = Cleaner.create();
segment.registerCleaner(cleaner);
// do some work
segment = null; // Cleaner might reclaim the segment memory now
```

Registering a segment with a cleaner does not prevent clients from calling `MemorySegment::close` explicitly. The API guarantees that the segment's cleanup action will be called at most once — either explicitly (by client code), or implicitly (by a cleaner). Since an unreachable segment cannot (by definition) be accessed by any thread, the cleaner can always release any memory resources associated with an unreachable segment regardless of whether it is a confined segment or a shared segment.

Alternatives

Keep using existing APIs such as `java.nio.ByteBuffer` or `sun.misc.Unsafe` or, worse, JNI.

Risks and Assumptions

Creating an API to access foreign memory in a way that is both safe and efficient is a daunting task. Since the spatial and temporal checks described in the previous sections need to be performed upon every access, it is crucial that JIT compilers be able to optimize away these checks by, e.g., hoisting them outside of hot loops. The JIT implementations will likely require some work to ensure that uses of the API are as efficient and optimizable as uses of existing APIs such as `ByteBuffer` and `Unsafe`.

Dependencies

The API described in this JEP will help the development of the native interoperation support that is a goal of Project Panama. This API can also be used to access non-volatile memory, already possible via JEP 352 (Non-Volatile Mapped Byte Buffers), in a more general and efficient way.