JEP 383: Foreign-Memory Access API (Second Incubator)

Owner	Maurizio Cimadamore
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	15
Component	core-libs
Discussion	panama dash dev at openjdk dot java dot net
Relates to	JEP 370: Foreign-Memory Access API (Incubator) JEP 393: Foreign-Memory Access API (Third Incubator)
Reviewed by	Paul Sandoz
Endorsed by	Mark Reinhold
Created	2020/04/10 15:41
Updated	2021/08/28 00:14
Issue	8242499

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 proposed by JEP 370 and targeted to Java 14 in late 2019 as an incubating API. This JEP proposes to incorporate refinements based on feedback, and re-incubate the API in Java 15. The following changes have been included as part of this API refresh:

- A rich `VarHandle` combinator API, to customize memory access var handles;
- Targeted support for parallel processing of a memory segment via the `Spliterator` interface;
- Enhanced support for *mapped* memory segments (e.g., `MappedMemorySegment::force`);
- Safe API points to support serial confinement (e.g., to transfer thread ownership between two threads); and
- Unsafe API points to manipulate and dereference addresses coming from, e.g., native calls, or to wrap such addresses into synthetic memory segments.

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.
- *Determinism:* Deallocation operations on foreign memory should be explicit in source code.
- *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. There are generally two kinds of addresses: A *checked* address is an offset within a given memory segment, while an *unchecked* address is an address whose spatial and temporal bounds are unknown, as in the case of a memory address obtained -- unsafely -- from native code.
- 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, memory segments are also *temporally bounded*, which means they must be created, used, and then closed when no longer in use. Closing a segment is always an explicit operation and 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 will result 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.

Dereferencing the memory associated with a segment is achieved by obtaining a *var handle*, which is 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 has an *access coordinate* of type `MemoryAddress` that serves as the address at which the 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)) {
    MemoryAddress base = segment.baseAddress();
    for (int i = 0; i < 25; i++) {
        intHandle.set(base.addOffset(i * 4), i);
    }
}
```

Memory-access var handles can acquire extra access coordinates, of type `long`, to support more complex addressing schemes, such as multi-dimensional addressing of an otherwise flat memory segment. Such memory-access var handles are

typically obtained by invoking combinator methods defined in the `MemoryHandles` class. For instance, a more direct way to set the elements of a native memory segment is through an *indexed* memory-access var handle, constructed as follows:

```
VarHandle intHandle = MemoryHandles.varHandle(int.class,
        ByteOrder.nativeOrder());
VarHandle indexedElementHandle = MemoryHandles.withStride(intHandle, 4);

try (MemorySegment segment = MemorySegment.allocateNative(100)) {
    MemoryAddress base = segment.baseAddress();
    for (int i = 0; i < 25; i++) {
        indexedElementHandle.set(base, (long) i, i);
    }
}
```

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)) {
    MemoryAddress base = segment.baseAddress();
    for (int i = 0; i < intArrayLayout.elementCount().getAsLong(); i++) {
        indexedElementHandle.set(base, (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.

Checked vs. unchecked addresses

Dereference operations are only possible on *checked* memory addresses. Checked addresses are typical in the API, such as the address obtained from a memory segment in the above code (`segment.baseAddress()`). However, if a memory address is *unchecked* and does not have any associated segment, then it cannot be dereferenced safely, since the runtime has no way to know the spatial and temporal bounds associated with the address. Examples of unchecked addresses are:

- the `NULL` address (MemoryAddress::NULL)
- address constructed from a long value (via the `MemoryAddress::ofLong` factory)

To dereference an unchecked address, a client has two options. If the address is known to fall within a memory segment the client already had, the client can perform a so called *rebase* operation (`MemoryAddress::rebase`), where the unchecked address' offset is re-interpreted relative to the segment's base address, yielding a new address instance which can be safely dereferenced. Alternatively, if no such segment exists, the client can create one unsafely, using the special `MemorySegment::ofNativeRestricted` 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, however, unsafe by its very nature, and must be used with care. For this reason, the Foreign Memory Access API only allow calls to this factory when the JDK 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.
- `debug` - like `permit`, but also dumps the stack corresponding to any given restricted.

We plan, in the future, to make access to restricted operations more integrated with the module system; that is, certain modules might *require* restricted native access; when an application which depends on said modules is executed, the user might need to provide *permissions* to said modules to perform restricted native operations, or the runtime will refuse to build the application's module graph.

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 on 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. If thread-confinement restrictions were to be removed, it would be possible for multiple threads to access and close the same segment concurrently - which might invalidate the safety guarantees provided by the Foreign Memory Access API, unless some very expensive form of locking was introduced to prevent access vs. close races.

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

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

This pattern of access is also known as *serial confinement* and might 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 into memory by the time the second thread accesses the segment.

Secondly, the contents of a memory segment can still be processed in *parallel* (e.g. using a framework such as Fork/Join) — by obtaining a `Spliterator` instance out of 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, PathElement.sequenceElement());

int sum = StreamSupport.stream(MemorySegment.spliterator(segment, seq_bulk), true)
                .mapToInt(slice -> {
                                        int res = 0;
                                MemoryAddress base = slice.baseAddress();
                                for (int i = 0; i < 100 ; i++) {
                                res += (int)intHandle.get(base, (long)i);
                                }
                                return res;
                }).sum();
```

The `MemorySegment::spliterator` takes a segment, a *sequence* layout and returns a spliterator instance which splits the segment into chunks which corresponds to the elements in the provided sequence layout. Here, we want to sum elements in an array which contains a million of elements; now, 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 of the original layouts, but where the elements are arranged in groups of 100 elements — which should 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. While the segment here is accessed from multiple threads concurrently, the access occurs in a regular fashion: a slice is created from the original segment, and given to a thread to perform some computation. The foreign memory access runtime knows if a thread is currently accessing a slice of the segment through the spliterator, and can hence enforce safety by *not* allowing a segment to be closed *while* parallel processing on same segment is taking place.

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 likely 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.