

Module `jdk.incubator.foreign`

Package `jdk.incubator.foreign`

`package jdk.incubator.foreign`

Classes to support low-level and efficient foreign memory/function access, directly from Java.

Foreign memory access

The key abstractions introduced to support foreign memory access are `MemorySegment` and `MemoryAddress`. The first models a contiguous memory region, which can reside either inside or outside the Java heap; the latter models an address - which also can reside either inside or outside the Java heap (and can sometimes be expressed as an offset into a given segment). A memory segment represents the main access coordinate of a memory access var handle, which can be obtained using the combinator methods defined in the `MemoryHandles` class; a set of common dereference operations is provided also by the `MemoryAccess` class, which can be useful for simple, non-structured access. Finally, the `MemoryLayout` class hierarchy enables description of *memory layouts* and basic operations such as computing the size in bytes of a given layout, obtain its alignment requirements, and so on. Memory layouts also provide an alternate, more abstract way, to produce memory access var handles, e.g. using *layout paths*. For example, to allocate an off-heap memory region big enough to hold 10 values of the primitive type `int`, and fill it with values ranging from 0 to 9, we can use the following code:

```
MemorySegment segment = MemorySegment.allocateNative(10 * 4, ResourceScope.newImplicitScope());
for (int i = 0 ; i < 10 ; i++) {
    MemoryAccess.setIntAtIndex(segment, i, 42);
}
```

Here create a *native* memory segment, that is, a memory segment backed by off-heap memory; the size of the segment is 40 bytes, enough to store 10 values of the primitive type `int`. Inside a loop, we then initialize the contents of the memory segment using the `MemoryAccess.setIntAtIndex(jdk.incubator.foreign.MemorySegment, long, int)` helper method; more specifically, if we view the memory segment as a set of 10 adjacent slots, `s[i]`, where `0 <= i < 10`, where the size of each slot is exactly 4 bytes, the initialization logic above will set each slot so that `s[i] = i`, again where `0 <= i < 10`.

Deterministic deallocation

When writing code that manipulates memory segments, especially if backed by memory which resides outside the Java heap, it is often crucial that the resources associated with a memory segment are released when the segment is no longer in use, and in a timely fashion. For this reason, there might be cases where waiting for the garbage collector to determine that a segment is `unreachable` is not optimal. Clients that operate under these assumptions might want to programmatically release the memory associated with a memory segment. This can be done, using the `ResourceScope` abstraction, as shown below:

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    MemorySegment segment = MemorySegment.allocateNative(10 * 4, scope);
    for (int i = 0 ; i < 10 ; i++) {
        MemoryAccess.setIntAtIndex(segment, i, 42);
    }
}
```

This example is almost identical to the prior one; this time we first create a so called *resource scope*, which is used to *bind* the life-cycle of the segment created immediately afterwards. Note the use of the *try-with-resources* construct: this idiom ensures that all the memory resources associated with the segment will be released at the end of the block, according to the semantics described in Section 14.20.3² of *The Java Language Specification*.

Safety

This API provides strong safety guarantees when it comes to memory access. First, when dereferencing a memory segment, the access coordinates are validated (upon access), to make sure that access does not occur at an address which resides *outside* the boundaries of the memory segment used by the dereference operation. We call this guarantee *spatial safety*; in other words, access to memory segments is bounds-checked, in the same way as array access is, as described in Section 15.10.4² of *The Java Language Specification*.

Since memory segments can be closed (see above), segments are also validated (upon access) to make sure that the resource scope associated with the segment being accessed has not been closed prematurely. We call this guarantee *temporal safety*. Together, spatial and temporal safety ensure that each memory access operation either succeeds - and accesses a valid memory location - or fails.

Foreign function access

The key abstractions introduced to support foreign function access are `SymbolLookup` and `CLinker`. The former is used to lookup symbols inside native libraries; the latter provides linking capabilities which allow to model foreign functions as `MethodHandle` instances, so that clients can perform foreign function calls directly in Java, without the need for intermediate layers of native code (as it's the case with the `Java Native Interface (JNI)`).

For example, to compute the length of a string using the C standard library function `strlen` on a Linux x64 platform, we can use the following code:

```
MethodHandle strlen = CLinker.getInstance().downcallHandle(
    CLinker.systemLookup().lookup("strlen").get(),
    MethodType.methodType(long.class, MemoryAddress.class),
    FunctionDescriptor.of(CLinker.C_LONG, CLinker.C_POINTER)
);

try (var scope = ResourceScope.newConfinedScope()) {
    var cString = CLinker.toCString("Hello", scope);
    long len = (long)strlen.invokeExact(cString.address()); // 5
}
```

Here, we lookup the `strlen` symbol in the `system lookup`. Then, we obtain a linker instance (see `CLinker.getInstance()`) and we use it to obtain a method handle which targets the `strlen` library symbol. To complete the linking successfully, we must provide (i) a `MethodType` instance, describing the type of the resulting method handle and (ii) a `FunctionDescriptor` instance, describing the signature of the `strlen` function. From this information, the linker will uniquely determine the sequence of steps which will turn the method handle invocation (here performed using `MethodHandle.invokeExact(java.lang.Object...)`) into a foreign function call, according to the rules specified by the platform C ABI.

The `CLinker` class also provides many useful methods for interacting with native code, such as converting Java strings into native strings and viceversa (see `CLinker.toString(java.lang.String, ResourceScope)` and `CLinker.toString(jdk.incubator.foreign.MemorySegment)`, respectively), as demonstrated in the above example.

Foreign addresses

When a memory segment is created from Java code, the segment properties (spatial bounds, temporal bounds and confinement) are fully known at segment creation. But when interacting with native libraries, clients will often receive *raw* pointers; such pointers have no spatial bounds (example: does the C type `char*` refer to a single `char` value, or an array of `char` values, of given size?), no notion of temporal bounds, nor thread-confinement.

When clients receive a `MemoryAddress` instance from a foreign function call, it might be necessary to obtain a `MemorySegment` instance to dereference the memory pointed to by that address. To do that, clients can proceed in three different ways, described below.

First, if the memory address is known to belong to a segment the client already owns, a *rebase* operation can be performed; in other words, the client can ask the address what its offset relative to a given segment is, and, then, proceed to dereference the original segment accordingly, as follows:

```
MemorySegment segment = MemorySegment.allocateNative(100, scope);
...
MemoryAddress addr = ... //obtain address from native code
int x = MemoryAccess.getIntAtOffset(segment, addr.segmentOffset(segment));
```

Secondly, if the client does *not* have a segment which contains a given memory address, it can create one *unsafely*, using the `MemoryAddress.asSegment(long, ResourceScope)` factory. This allows the client to inject extra knowledge about spatial bounds which might, for instance, be available in the documentation of the foreign function which produced the native address. Here is how an unsafe segment can be created from a native address:

```
ResourceScope scope = ... // initialize a resource scope object
MemoryAddress addr = ... //obtain address from native code
MemorySegment segment = addr.asSegment(4, scope); // segment is 4 bytes long
int x = MemoryAccess.getInt(segment);
```

Alternatively, the client can fall back to use the so called *everything* segment - that is, a primordial segment which covers the entire native heap. This segment can be obtained by calling the `MemorySegment.globalNativeSegment()` method, so that dereference can happen without the need of creating any additional segment instances:

```
MemoryAddress addr = ... //obtain address from native code
int x = MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(), addr.toRawLongValue());
```

Upcalls

The `CLinker` interface also allows to turn an existing method handle (which might point to a Java method) into a native memory address (see `MemoryAddress`), so that Java code can effectively be passed to other foreign functions. For instance, we can write a method that compares two integer values, as follows:

```
class IntComparator {
    static int intCompare(MemoryAddress addr1, MemoryAddress addr2) {
        return MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(), addr1.toRawLongValue()) -
            MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(), addr2.toRawLongValue());
    }
}
```

The above method dereferences two memory addresses containing an integer value, and performs a simple comparison by returning the difference between such values. We can then obtain a method handle which targets the above static method, as follows:

```
MethodHandle intCompareHandle = MethodHandles.lookup().findStatic(IntComparator.class,
    "intCompare",
    MethodType.methodType(int.class, MemoryAddress.class, MemoryAddress.class));
```

Now that we have a method handle instance, we can link it into a fresh native memory address, using the `CLinker` interface, as follows:

```
ResourceScope scope = ...
MemoryAddress comparFunc = CLinker.getInstance().upcallStub(
    intCompareHandle,
    FunctionDescriptor.of(C_INT, C_POINTER, C_POINTER),
    scope
);
```

As before, we need to provide a `FunctionDescriptor` instance describing the signature of the function pointer we want to create; as before, this, coupled with the method handle type, uniquely determines the sequence of steps which will allow foreign code to call `intCompareHandle` according to the rules specified by the platform C ABI. The lifecycle of the memory address returned by `CLinker.upcallStub(java.lang.invoke.MethodHandle, jdk.incubator.foreign.FunctionDescriptor, jdk.incubator.foreign.ResourceScope)` is tied to the `resource scope` parameter passed to that method.

Restricted methods

Some methods in this package are considered *restricted*. Restricted methods are typically used to bind native foreign data and/or functions to first-class Java API elements which can then be used directly by clients. For instance the restricted method `MemoryAddress.asSegment(long, ResourceScope)` can be used to create a fresh segment with given spatial bounds out of a native address.

Binding foreign data and/or functions is generally unsafe and, if done incorrectly, can result in VM crashes, or memory corruption when the bound Java API element is accessed. For instance, in the case of `MemoryAddress.asSegment(long, ResourceScope)`, if the provided spatial bounds are incorrect, a client of the segment returned by that method might crash the VM, or corrupt memory when attempting to dereference

said segment. For these reasons, it is crucial for code that calls a restricted method to never pass arguments that might cause incorrect binding of foreign data and/or functions to a Java API.

Access to restricted methods is *disabled* by default; to enable restricted methods, the command line option `--enable-native-access` must mention the name of the caller's module.

All Classes and Interfaces	Interfaces	Classes	Enum Classes
Class	Description		
Addressable	Represents a type which is <i>addressable</i> .		
CLinker	A C linker implements the C Application Binary Interface (ABI) calling conventions.		
CLinker.TypeKind	A C type kind.		
CLinker.ValList	An interface that models a C <code>va_list</code> .		
CLinker.ValList.Builder	A builder interface used to construct a C <code>va_list</code> .		
FunctionDescriptor	A function descriptor is made up of zero or more argument layouts and zero or one return layout.		
GroupLayout	A group layout is used to combine together multiple <i>member layouts</i> .		
MemoryAccess	This class defines ready-made static accessors which can be used to dereference memory segments in many ways.		
MemoryAddress	A memory address models a reference into a memory location.		
MemoryHandles	This class defines several factory methods for constructing and combining memory access var handles.		
MemoryLayout	A memory layout can be used to describe the contents of a memory segment in a <i>language neutral</i> fashion.		
MemoryLayout.PathElement	Instances of this class are used to form <i>layout paths</i> .		
MemoryLayouts	This class defines useful layout constants.		
MemorySegment	A memory segment models a contiguous region of memory.		
ResourceScope	A resource scope manages the lifecycle of one or more resources.		
ResourceScope.Handle	An abstraction modelling a resource scope handle.		
SegmentAllocator	This interface models a memory allocator.		
SequenceLayout	A sequence layout.		
SymbolLookup	A symbol lookup.		
ValueLayout	A value layout.		

[Report a bug or suggest an enhancement](#)

For further API reference and developer documentation see the [Java SE Documentation](#), which contains more detailed, developer-targeted descriptions with conceptual overviews, definitions of terms, workarounds, and working code examples. [Other versions](#).

Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.

Copyright © 1993, 2023, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.

All rights reserved. Use is subject to [license terms](#) and the [documentation redistribution policy](#). [Modify Cookie Preferences](#). [Modify Ad Choices](#).