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 main abstraction introduced to support foreign memory access is `MemorySegment`, which models a contiguous memory region, which can reside either inside or outside the Java heap. 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 and copy operations is provided also by the `MemorySegment` 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++) {
    segment.setAtIndex(ValueLayout.JAVA_INT, i, i);
}
```

This code creates 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; note how the dereference method accepts a value layout, which specifies the size, alignment constraints, byte order as well as the Java type (`int`, in this case) associated with the dereference operation. 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++) {
        segment.setAtIndex(ValueLayout.JAVA_INT, i, i);
    }
}
```

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 14.20.3 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 any 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 15.10.4 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`, `MemoryAddress` and `CLinker`. The first is used to look up symbols inside native libraries; the second is used to model native addresses (more on that later), while the third provides linking capabilities which allows modelling 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 is 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:

```
var linker = CLinker.systemCLinker();
MethodHandle strlen = linker.downcallHandle(
    linker.lookup("strlen").get(),
    FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS)
);

try (var scope = ResourceScope.newConfinedScope()) {
    var cString = MemorySegment.allocateNative(5 + 1, scope);
```

```
    cString.setUtf8String("Hello");
    long len = (long)strlen.invoke(cString); // 5
}
```

Here, we obtain a linker instance and we use it to look up the `strlen` symbol in the standard C library; a *downcall method handle* targeting said symbol is subsequently obtained. To complete the linking successfully, we must provide 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.invoke(java.lang.Object...)`) into a foreign function call, according to the rules specified by the platform C ABI. The `MemorySegment` class also provides many useful methods for interacting with native code, such as converting Java strings into native strings and back, 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. For example, the C type `char*` can refer to a single `char` value, or an array of `char` values, of given size. Nor do said pointers have any notion of temporal bounds or thread-confinement.

Raw pointers are modelled using the `MemoryAddress` class. When clients receive a memory address instance from a foreign function call, they can perform memory dereference on it directly, using one of the many *unsafe* dereference methods provided:

```
MemoryAddress addr = ... //obtain address from native code
int x = addr.get(ValueLayout.JAVA_INT, 0);
```

Alternatively, the client can create a memory segment *unsafely*. 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 = MemorySegment.ofAddress(addr, 4, scope); // segment is 4 bytes long
int x = segment.get(ValueLayout.JAVA_INT, 0);
```

Upcalls

The `CLinker` interface also allows clients to turn an existing method handle (which might point to a Java method) into a memory address, 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 addr1.get(ValueLayout.JAVA_INT, 0) - addr2.get(ValueLayout.JAVA_INT, 0);
    }
}
```

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:

```
FunctionDescriptor intCompareDescriptor = FunctionDescriptor.of(ValueLayout.JAVA_INT, ValueLayout.ADDRESS, Va
MethodHandle intCompareHandle = MethodHandles.lookup().findStatic(IntComparator.class,
                                                  "intCompare",
                                                  CLinker.upcallType(comparFunction));
```

As before, we need to create a `FunctionDescriptor` instance, this time describing the signature of the function pointer we want to create. The descriptor can be used to derive a method type that can be used to look up the method handle for `IntComparator.intCompare`.

Now that we have a method handle instance, we can turn it into a fresh function pointer, using the `CLinker` interface, as follows:

```
ResourceScope scope = ...
Addressable comparFunc = CLinker.systemCLinker().upcallStub(
    intCompareHandle, intCompareDescriptor, scope);
);
```

The `FunctionDescriptor` instance created in the previous step is then used to create a new upcall stub; the layouts in the function descriptors allow the linker to determine the sequence of steps which allow foreign code to call the stub for `intCompareHandle` according to the rules specified by the platform C ABI. The lifecycle of the upcall stub returned by is tied to the resource scope provided when the upcall stub is created. This same scope is made available by the `NativeSymbol` instance returned by 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 `MemorySegment.ofAddress(MemoryAddress, 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 `MemorySegment.ofAddress(MemoryAddress, 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**

Class	Description
Addressable	Represents a type which is *addressable*.
CLinker	A C linker implements the C Application Binary Interface (ABI) calling conventions.
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 multiple *member layouts*.
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.
MemoryLayout.PathElement	Instances of this class are used to form *layout paths*.
MemorySegment	A memory segment models a contiguous region of memory.
NativeSymbol	A native symbol models a reference to a location (typically the entry point of a function) in a native library.
ResourceScope	A resource scope manages the lifecycle of one or more resources.
SegmentAllocator	This interface models a memory allocator.
SequenceLayout	A sequence layout.
SymbolLookup	A symbol lookup.
VaList	An interface that models a variable argument list, similar in functionality to a C `va_list`.
VaList.Builder	A builder interface used to construct a variable argument list.
ValueLayout	A value layout.
ValueLayout.OfAddress	A value layout whose carrier is `MemoryAddress.class`.
ValueLayout.OfBoolean	A value layout whose carrier is `boolean.class`.
ValueLayout.OfByte	A value layout whose carrier is `byte.class`.
ValueLayout.OfChar	A value layout whose carrier is `char.class`.
ValueLayout.OfDouble	A value layout whose carrier is `double.class`.
ValueLayout.OfFloat	A value layout whose carrier is `float.class`.
ValueLayout.OfInt	A value layout whose carrier is `int.class`.
ValueLayout.OfLong	A value layout whose carrier is `long.class`.
ValueLayout.OfShort	A value layout whose carrier is `short.class`.