Module java.base

Package java.lang.foreign

`package java.lang.foreign`

Provides low-level access to memory and functions outside the Java runtime.

Foreign memory access

The main abstraction introduced to support foreign memory access is `MemorySegment`[PREVIEW], which models a contiguous region of memory, residing either inside or outside the Java heap. The contents of a memory segment can be described using a `memory layout`[PREVIEW], which provides basic operations to query sizes, offsets and alignment constraints. Memory layouts also provide an alternate, more abstract way, to access memory segments using var handles[PREVIEW], which can be computed using *layout paths*. For example, to allocate an off-heap region of memory 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, SegmentScope.auto());
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`. The segment is associated with an automatic scope[PREVIEW]. This means that the off-heap region of memory backing the segment is managed, automatically, by the garbage collector. As such, the off-heap memory backing the native segment will be released at some unspecified point *after* the segment becomes unreachable. This is similar to what happens with direct buffers created via `ByteBuffer.allocateDirect(int)`. It is also possible to manage the lifecycle of allocated native segments more directly, as shown in a later section.

Inside a loop, we then initialize the contents of the memory segment; note how the access method[PREVIEW] accepts a value layout[PREVIEW], which specifies the size, alignment constraint, byte order as well as the Java type (`int`, in this case) associated with the access 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 backing a memory segment. This can be done, using the `Arena`[PREVIEW] abstraction, as shown below:

```
try (Arena arena = Arena.openConfined()) {
    MemorySegment segment = arena.allocate(10 * 4);
    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 an arena which is used to allocate multiple native segments which share the same life-cycle. That is, all the segments allocated by the arena will be associated with the same scope[PREVIEW]. Note the use of the *try-with-resources* construct: this idiom ensures that the off-heap region of memory backing the native 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 access 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 created with an arena can become invalid (see above), segments are also validated (upon access) to make sure that the scope associated with the segment being accessed is still alive. We call this guarantee *temporal safety*. Together, spatial and temporal safety ensure that each memory access operation either succeeds - and accesses a valid location within the region of memory backing the memory segment - or fails.

Foreign function access

The key abstractions introduced to support foreign function access are `SymbolLookup`[PREVIEW], `FunctionDescriptor`[PREVIEW] and `Linker`[PREVIEW]. The first is used to look up symbols inside libraries; the second is used to model the signature of foreign functions, 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 C/C++ 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:

```
Linker linker = Linker.nativeLinker();
SymbolLookup stdlib = linker.defaultLookup();
MethodHandle strlen = linker.downcallHandle(
    stdlib.find("strlen").get(),
```

```
    FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS)
);

try (Arena arena = Arena.openConfined()) {
    MemorySegment cString = arena.allocateUtf8String("Hello");
    long len = (long)strlen.invoke(cString); // 5
}
```

Here, we obtain a native linker[PREVIEW] and we use it to look up[PREVIEW] the `strlen` symbol in the standard C library; a *downcall method handle* targeting said symbol is subsequently obtained[PREVIEW]. To complete the linking successfully, we must provide a `FunctionDescriptor`[PREVIEW] 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 ABI of the underlying platform. The `Arena`[PREVIEW] class also provides many useful methods for interacting with foreign code, such as converting[PREVIEW] Java strings into zero-terminated, UTF-8 strings, as demonstrated in the above example.

Upcalls

The `Linker`[PREVIEW] interface also allows clients to turn an existing method handle (which might point to a Java method) into a memory segment, 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(MemorySegment addr1, MemorySegment addr2) {
        return addr1.get(ValueLayout.JAVA_INT, 0) -
               addr2.get(ValueLayout.JAVA_INT, 0);

    }
}
```

The above method accesses two foreign memory segments 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.asUnbounded(),
                                                                ValueLayout.ADDRESS.asUnbounded());
MethodHandle intCompareHandle = MethodHandles.lookup().findStatic(IntComparator.class,
                                                  "intCompare",
                                                  intCompareDescriptor.toMethodType());
```

As before, we need to create a `FunctionDescriptor`[PREVIEW] instance, this time describing the signature of the function pointer we want to create. The descriptor can be used to derive[PREVIEW] 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 `Linker`[PREVIEW] interface, as follows:

```
SegmentScope scope = ...
MemorySegment comparFunc = Linker.nativeLinker().upcallStub(
    intCompareHandle, intCompareDescriptor, scope);
);
```

The `FunctionDescriptor`[PREVIEW] instance created in the previous step is then used to create[PREVIEW] 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 ABI of the underlying platform. The lifecycle of the upcall stub is tied to the scope[PREVIEW] provided when the upcall stub is created. This same scope is made available by the `MemorySegment`[PREVIEW] 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(long, long, SegmentScope)`[PREVIEW] can be used to create a fresh segment with the 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(long, long, SegmentScope)`[PREVIEW], 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 access 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.

Given the potential danger of restricted methods, the Java runtime issues a warning on the standard error stream every time a restricted method is invoked. Such warnings can be disabled by granting access to restricted methods to selected modules. This can be done either via implementation-specific command line options, or programmatically, e.g. by calling `ModuleLayer.Controller.enableNativeAccess(java.lang.Module)`[PREVIEW].

For every class in this package, unless specified otherwise, any method arguments of reference type must not be null, and any null argument will elicit a `NullPointerException`. This fact is not individually documented for methods of this API.

API Note:

Usual memory model guarantees, for example stated in 6.6 and 10.4, do not apply when accessing native memory segments as these segments are backed by off-heap regions of memory.

Implementation Note:

In the reference implementation, access to restricted methods can be granted to specific modules using the command line option `--enable-native-access=M1,M2, ... Mn`, where `M1`, `M2`, `... Mn` are module names (for the unnamed module, the special value `ALL-UNNAMED` can be used). If this option is specified, access to restricted methods is only granted to the modules listed by that option. If this option is not specified, access to restricted methods is enabled for all modules, but access to restricted methods will result in runtime warnings.

Related Packages

Package	Description
java.lang	Provides classes that are fundamental to the design of the Java programming language.

Interfaces

Class	Description
Arena[PREVIEW]	**Preview.** An arena controls the lifecycle of memory segments, providing both flexible allocation and timely deallocation.
FunctionDescriptor[PREVIEW]	**Preview.** A function descriptor models the signature of foreign functions.
GroupLayout[PREVIEW]	**Preview.** A compound layout that aggregates multiple *member layouts*.
Linker[PREVIEW]	**Preview.** A linker provides access to foreign functions from Java code, and access to Java code from foreign functions.
Linker.Option[PREVIEW]	**Preview.** A linker option is used to indicate additional linking requirements to the linker, besides what is described by a function descriptor.
Linker.Option.CaptureCallState[PREVIEW]	**Preview.** A linker option for saving portions of the execution state immediately after calling a foreign function associated with a downcall method handle, before it can be overwritten by the runtime, or read through conventional means.
MemoryLayout[PREVIEW]	**Preview.** A memory layout describes the contents of a memory segment.
MemoryLayout.PathElement[PREVIEW]	**Preview.** An element in a *layout path*.
MemorySegment[PREVIEW]	**Preview.** A memory segment provides access to a contiguous region of memory.
PaddingLayout[PREVIEW]	**Preview.** A padding layout.
SegmentAllocator[PREVIEW]	**Preview.** An object that may be used to allocate memory segments[PREVIEW].
SegmentScope[PREVIEW]	**Preview.** A segment scope controls access to memory segments.
SequenceLayout[PREVIEW]	**Preview.** A compound layout that denotes a repetition of a given *element layout*.
StructLayout[PREVIEW]	**Preview.** A group layout whose member layouts are laid out one after the other.
SymbolLookup[PREVIEW]	**Preview.** A *symbol lookup* retrieves the address of a symbol in one or more libraries.
UnionLayout[PREVIEW]	**Preview.** A group layout whose member layouts are laid out at the same starting offset.
VaList[PREVIEW]	**Preview.** Helper class to create and manipulate variable argument lists, similar in functionality to a C `va_list`.
VaList.Builder[PREVIEW]	**Preview.** A builder used to construct a variable argument list[PREVIEW].
ValueLayout[PREVIEW]	**Preview.** A layout that models values of basic data types.

ValueLayout.OfAddress[PREVIEW]	**Preview.** A value layout whose carrier is `MemorySegment.class`.
ValueLayout.OfBoolean[PREVIEW]	**Preview.** A value layout whose carrier is `boolean.class`.
ValueLayout.OfByte[PREVIEW]	**Preview.** A value layout whose carrier is `byte.class`.
ValueLayout.OfChar[PREVIEW]	**Preview.** A value layout whose carrier is `char.class`.
ValueLayout.OfDouble[PREVIEW]	**Preview.** A value layout whose carrier is `double.class`.
ValueLayout.OfFloat[PREVIEW]	**Preview.** A value layout whose carrier is `float.class`.
ValueLayout.OfInt[PREVIEW]	**Preview.** A value layout whose carrier is `int.class`.
ValueLayout.OfLong[PREVIEW]	**Preview.** A value layout whose carrier is `long.class`.
ValueLayout.OfShort[PREVIEW]	**Preview.** A value layout whose carrier is `short.class`.