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 memory region, 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 dereference memory segments using access var handlesPREVIEW, which can be computed 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, MemorySession.openImplicit());
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 methodPREVIEW accepts a value layoutPREVIEW, 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 `MemorySession`PREVIEW abstraction, as shown below:

```
try (MemorySession session = MemorySession.openConfined()) {
    MemorySegment segment = MemorySegment.allocateNative(10 * 4, session);
    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 *memory session*, 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 memory session 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`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.lookup("strlen").get(),
    FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS)
);

try (MemorySession session = MemorySession.openConfined()) {
    MemorySegment cString = MemorySegment.allocateNative(5 + 1, session);
    cString.setUtf8String(0, "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 `MemorySegment`[PREVIEW] class also provides many useful methods for interacting with foreign code, such as converting Java strings into[PREVIEW] zero-terminated, UTF-8 strings and back[PREVIEW], 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 foreign functions, 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`[PREVIEW] 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[PREVIEW] provided:

```
MemoryAddress addr = ... // obtain address from foreign function call
int x = addr.get(ValueLayout.JAVA_INT, 0);
```

Alternatively, the client can create[PREVIEW] 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 memory address:

```
MemorySession session = ... // initialize a memory session object
MemoryAddress addr = ... // obtain address from foreign function call
MemorySegment segment = MemorySegment.ofAddress(addr, 4, session); // segment is 4 bytes long
int x = segment.get(ValueLayout.JAVA_INT, 0);
```

Upcalls

The `Linker`[PREVIEW] 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, Val
MethodHandle intCompareHandle = MethodHandles.lookup().findStatic(IntComparator.class,
                                                "intCompare",
                                                Linker.upcallType(comparFunction));
```

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:

```
MemorySession session = ...
Addressable comparFunc = Linker.nativeLinker().upcallStub(
    intCompareHandle, intCompareDescriptor, session);
);
```

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 memory session[PREVIEW] provided when the upcall stub is created. This same session 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(MemoryAddress, long, MemorySession)`[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(MemoryAddress, long, MemorySession)`[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 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 can be controlled 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.

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.

Related Packages

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

All Classes and Interfaces | **Interfaces** | **Classes**

Class	Description
AddressablePREVIEW	**Preview.** An object that may be projected down to a memory addressPREVIEW.
FunctionDescriptorPREVIEW	**Preview.** A function descriptor is made up of zero or more argument layouts and zero or one return layout.
GroupLayoutPREVIEW	**Preview.** A compound layout that aggregates multiple *member layouts*.
LinkerPREVIEW	**Preview.** A linker provides access to foreign functions from Java code, and access to Java code from foreign functions.
MemoryAddressPREVIEW	**Preview.** A memory address models a reference into a memory location.
MemoryLayoutPREVIEW	**Preview.** A memory layout can be used to describe the contents of a memory segment.
MemoryLayout.PathElementPREVIEW	**Preview.** An element in a *layout path*.
MemorySegmentPREVIEW	**Preview.** A memory segment models a contiguous region of memory.
MemorySessionPREVIEW	**Preview.** A memory session manages the lifecycle of one or more resources.
SegmentAllocatorPREVIEW	**Preview.** An object that may be used to allocate memory segmentsPREVIEW.
SequenceLayoutPREVIEW	**Preview.** A compound layout that denotes a repetition of a given *element layout*.
SymbolLookupPREVIEW	**Preview.** A *symbol lookup* is an object that may be used to retrieve the address of a symbol in one or more libraries.
VaListPREVIEW	**Preview.** A variable argument list, similar in functionality to a C `va_list`.
VaList.BuilderPREVIEW	**Preview.** A builder used to construct a variable argument listPREVIEW.
ValueLayoutPREVIEW	**Preview.** A value layout.
ValueLayout.OfAddressPREVIEW	**Preview.** A value layout whose carrier is `MemoryAddress.class`.
ValueLayout.OfBooleanPREVIEW	**Preview.** A value layout whose carrier is `boolean.class`.
ValueLayout.OfBytePREVIEW	**Preview.** A value layout whose carrier is `byte.class`.
ValueLayout.OfCharPREVIEW	**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`.

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.