Module java.base
Package java.lang.foreign

Interface Linker

```
public sealed interface Linker
```

> **`Linker` is a preview API of the Java platform.**
> *Programs can only use `Linker` when preview features are enabled.*
> *Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.*

A linker provides access to foreign functions from Java code, and access to Java code from foreign functions.

Foreign functions typically reside in libraries that can be loaded on-demand. Each library conforms to a specific ABI (Application Binary Interface). An ABI is a set of calling conventions and data types associated with the compiler, OS, and processor where the library was built. For example, a C compiler on Linux/x64 usually builds libraries that conform to the SystemV ABI.

A linker has detailed knowledge of the calling conventions and data types used by a specific ABI. For any library which conforms to that ABI, the linker can mediate between Java code running in the JVM and foreign functions in the library. In particular:

- A linker allows Java code to link against foreign functions, via downcall method handles; and
- A linker allows foreign functions to call Java method handles, via the generation of upcall stubs.

In addition, a linker provides a way to look up foreign functions in libraries that conform to the ABI. Each linker chooses a set of libraries that are commonly used on the OS and processor combination associated with the ABI. For example, a linker for Linux/x64 might choose two libraries: `libc` and `libm`. The functions in these libraries are exposed via a symbol lookup.

The `nativeLinker()` method provides a linker for the ABI associated with the OS and processor where the Java runtime is currently executing. This linker also provides access, via its default lookup, to the native libraries loaded with the Java runtime.

Downcall method handles

Linking a foreign function is a process which requires a function descriptor, a set of memory layouts which, together, specify the signature of the foreign function to be linked, and returns, when complete, a downcall method handle, that is, a method handle that can be used to invoke the target foreign function.

The Java method type associated with the returned method handle is derived from the argument and return layouts in the function descriptor. More specifically, given each layout `L` in the function descriptor, a corresponding carrier `C` is inferred, as described below:

- if `L` is a `ValueLayout`[PREVIEW] with carrier `E` then there are two cases:
  - if `L` occurs in a parameter position and `E` is `MemoryAddress.class`, then `C = Addressable.class`;
  - otherwise, `C = E`;
- or, if `L` is a `GroupLayout`[PREVIEW], then `C` is set to `MemorySegment.class`

The downcall method handle type, derived as above, might be decorated by additional leading parameters, in the given order if both are present:

- If the downcall method handle is created without specifying a target address, the downcall method handle type features a leading parameter of type `Addressable`[PREVIEW], from which the address of the target foreign function can be derived.
- If the function descriptor's return layout is a group layout, the resulting downcall method handle accepts an additional leading parameter of type `SegmentAllocator`[PREVIEW], which is used by the linker runtime to allocate the memory region associated with the struct returned by the downcall method handle.

Upcall stubs

Creating an upcall stub requires a method handle and a function descriptor; in this case, the set of memory layouts in the function descriptor specify the signature of the function pointer associated with the upcall stub.

The type of the provided method handle has to match the Java method type associated with the upcall stub, which is derived from the argument and return layouts in the function descriptor. More specifically, given each layout `L` in the function descriptor, a corresponding carrier `C` is inferred, as described below:

- If `L` is a `ValueLayout`[PREVIEW] with carrier `E` then there are two cases:
  - If `L` occurs in a return position and `E` is `MemoryAddress.class`, then `C = Addressable.class`;
  - Otherwise, `C = E`;
- Or, if `L` is a `GroupLayout`[PREVIEW], then `C` is set to `MemorySegment.class`

Upcall stubs are modelled by instances of type `MemorySegment`[PREVIEW]; upcall stubs can be passed by reference to other downcall method handles (as `MemorySegment`[PREVIEW] implements the `Addressable`[PREVIEW] interface) and, when no longer required, they can be released[PREVIEW], via their associated session[PREVIEW].

Safety considerations

Creating a downcall method handle is intrinsically unsafe. A symbol in a foreign library does not, in general, contain enough signature information (e.g. arity and types of foreign function parameters). As a consequence, the linker runtime cannot validate linkage requests. When a client interacts with a downcall method handle obtained through an invalid linkage request (e.g. by specifying a function descriptor featuring too many argument layouts), the result of such interaction is unspecified and can lead to JVM crashes. On downcall handle invocation, the linker runtime guarantees the following for any argument that is a memory resource `R` (of type `MemorySegment`[PREVIEW] or `VaList`[PREVIEW]):

- The memory session of R is alive[PREVIEW]. Otherwise, the invocation throws `IllegalStateException`;
- The invocation occurs in same thread as the one owning[PREVIEW] the memory session of R, if said session is confined. Otherwise, the invocation throws `WrongThreadException`; and
- The memory session of R is kept alive[PREVIEW] (and cannot be closed) during the invocation.

When creating upcall stubs the linker runtime validates the type of the target method handle against the provided function descriptor and report an error if any mismatch is detected. As for downcalls, JVM crashes might occur, if the foreign code casts the function pointer associated with an upcall stub to a type that is incompatible with the provided function descriptor. Moreover, if the target method handle associated with an upcall stub returns a memory address[PREVIEW], clients must ensure that this address cannot become invalid after the upcall completes. This can lead to unspecified behavior, and even JVM crashes, since an upcall is typically executed in the context of a downcall method handle invocation.

Implementation Requirements:

Implementations of this interface are immutable, thread-safe and value-based.

Since:

19

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
`SymbolLookup`[PREVIEW]	`defaultLookup()`	Returns a symbol lookup for symbols in a set of commonly used libraries.
default `MethodHandle`	`downcallHandle(Addressable`[PREVIEW] `symbol, FunctionDescriptor`[PREVIEW] `function)`	Creates a method handle which can be used to call a target foreign function with the given signature and address.
`MethodHandle`	`downcallHandle (FunctionDescriptor`[PREVIEW] `function)`	Creates a method handle which can be used to call a target foreign function with the given signature.
static `MethodType`	`downcallType (FunctionDescriptor`[PREVIEW] `functionDescriptor)`	Returns the downcall method handle type associated with the given function descriptor.
static `Linker`[PREVIEW]	`nativeLinker()`	Returns a linker for the ABI associated with the underlying native platform.
`MemorySegment`[PREVIEW]	`upcallStub(MethodHandle target, FunctionDescriptor`[PREVIEW] `function, MemorySession`[PREVIEW] `session)`	Creates a stub which can be passed to other foreign functions as a function pointer, with the given memory session.
static `MethodType`	`upcallType (FunctionDescriptor`[PREVIEW] `functionDescriptor)`	Returns the method handle type associated with an upcall stub with the given function descriptor.

Method Details

nativeLinker

`static Linker`[PREVIEW] `nativeLinker()`

Returns a linker for the ABI associated with the underlying native platform. The underlying native platform is the combination of OS and processor where the Java runtime is currently executing.

When interacting with the returned linker, clients must describe the signature of a foreign function using a `function descriptor`[PREVIEW] whose argument and return layouts are specified as follows:

- Scalar types are modelled by a value layout[PREVIEW] instance of a suitable carrier. Example of scalar types in C are `int`, `long`, `size_t`, etc. The mapping between a scalar type and its corresponding layout is dependent on the ABI of the returned linker;
- Composite types are modelled by a group layout[PREVIEW]. Depending on the ABI of the returned linker, additional padding[PREVIEW] member layouts might be required to conform to the size and alignment constraints of a composite type definition in C (e.g. using `struct` or `union`); and
- Pointer types are modelled by a value layout[PREVIEW] instance with carrier `MemoryAddress`[PREVIEW]. Examples of pointer types in C are `int**` and `int(*)(size_t*, size_t*);`

Any layout not listed above is *unsupported*; function descriptors containing unsupported layouts will cause an `IllegalArgumentException` to be thrown, when used to create a `downcall method handle` or an upcall stub.

Variadic functions (e.g. a C function declared with a trailing ellipses `...` at the end of the formal parameter list or with an empty formal parameter list) are not supported directly. However, it is possible to link a variadic function by using a *variadic*[PREVIEW] function descriptor, in which the specialized signature of a given variable arity callsite is described in full. Alternatively, where the foreign library allows it, clients might be able to interact with variadic functions by passing a trailing parameter of type `VaList`[PREVIEW] (e.g. as in `vsprintf`).

This method is *restricted*. Restricted methods are unsafe, and, if used incorrectly, their use might crash the JVM or, worse, silently result in memory corruption. Thus, clients should refrain from depending on restricted methods, and use safe and supported functionalities, where possible.

API Note:

It is not currently possible to obtain a linker for a different combination of OS and processor.

Implementation Note:

The libraries exposed by the default lookup associated with the returned linker are the native libraries loaded in the process where the Java runtime is currently executing. For example, on Linux, these libraries typically include `libc`, `libm` and `libdl`.

Returns:

a linker for the ABI associated with the OS and processor where the Java runtime is currently executing.

Throws:

`UnsupportedOperationException` - if the underlying native platform is not supported.

`IllegalCallerException` - if access to this method occurs from a module `M` and the command line option `--enable-native-access` is specified, but does not mention the module name `M`, or `ALL-UNNAMED` in case `M` is an unnamed module.

downcallHandle

```
default MethodHandle downcallHandle(Addressable[PREVIEW] symbol,
                                    FunctionDescriptor[PREVIEW] function)
```

Creates a method handle which can be used to call a target foreign function with the given signature and address.

If the provided method type's return type is `MemorySegment`, then the resulting method handle features an additional prefix parameter, of type `SegmentAllocator`[PREVIEW], which will be used by the linker runtime to allocate structs returned by-value.

Calling this method is equivalent to the following code:

```
linker.downcallHandle(function).bindTo(symbol);
```

Parameters:

`symbol` - the address of the target function.

`function` - the function descriptor of the target function.

Returns:

a downcall method handle. The method handle type is *inferred*

Throws:

`IllegalArgumentException` - if the provided function descriptor is not supported by this linker. or if the symbol is `MemoryAddress.NULL`[PREVIEW]

downcallHandle

```
MethodHandle downcallHandle(FunctionDescriptor[PREVIEW] function)
```

Creates a method handle which can be used to call a target foreign function with the given signature. The resulting method handle features a prefix parameter (as the first parameter) corresponding to the foreign function entry point, of type `Addressable`[PREVIEW], which is used to specify the address of the target function to be called.

If the provided function descriptor's return layout is a `GroupLayout`[PREVIEW], then the resulting method handle features an additional prefix parameter (inserted immediately after the address parameter), of type `SegmentAllocator`[PREVIEW]), which will be used by the linker runtime to allocate structs returned by-value.

The returned method handle will throw an `IllegalArgumentException` if the `Addressable`[PREVIEW] parameter passed to it is associated with the `MemoryAddress.NULL`[PREVIEW] address, or a `NullPointerException` if that parameter is `null`.

Parameters:

`function` - the function descriptor of the target function.

Returns:

a downcall method handle. The method handle type is *inferred* from the provided function descriptor.

Throws:

`IllegalArgumentException` - if the provided function descriptor is not supported by this linker.

upcallStub

```
MemorySegment[PREVIEW] upcallStub(MethodHandle target,
                                  FunctionDescriptor[PREVIEW] function,
                                  MemorySession[PREVIEW] session)
```

Creates a stub which can be passed to other foreign functions as a function pointer, with the given memory session. Calling such a function pointer from foreign code will result in the execution of the provided method handle.

The returned memory segment's base address points to the newly allocated upcall stub, and is associated with the provided memory session. When such session is closed, the corresponding upcall stub will be deallocated.

The target method handle should not throw any exceptions. If the target method handle does throw an exception, the VM will exit with a non-zero exit code. To avoid the VM aborting due to an uncaught exception, clients could wrap all code in the target method handle in a try/catch block that catches any `Throwable`, for instance by using the `MethodHandles.catchException(MethodHandle, Class, MethodHandle)` method handle combinator, and handle exceptions as desired in the corresponding catch block.

Parameters:

`target` - the target method handle.

`function` - the upcall stub function descriptor.

`session` - the upcall stub memory session.

Returns:

a zero-length segment whose base address is the address of the upcall stub.

Throws:

`IllegalArgumentException` - if the provided function descriptor is not supported by this linker.

`IllegalArgumentException` - if it is determined that the target method handle can throw an exception, or if the target method handle has a type that does not match the upcall stub *inferred type*.

`IllegalStateException` - if `session` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning[PREVIEW] `session`.

defaultLookup

`SymbolLookup`[PREVIEW] `defaultLookup()`

Returns a symbol lookup for symbols in a set of commonly used libraries.

Each `Linker`[PREVIEW] is responsible for choosing libraries that are widely recognized as useful on the OS and processor combination supported by the `Linker`[PREVIEW]. Accordingly, the precise set of symbols exposed by the symbol lookup is unspecified; it varies from one `Linker`[PREVIEW] to another.

Implementation Note:

It is strongly recommended that the result of `defaultLookup()` exposes a set of symbols that is stable over time. Clients of `defaultLookup()` are likely to fail if a symbol that was previously exposed by the symbol lookup is no longer exposed.

If an implementer provides `Linker`[PREVIEW] implementations for multiple OS and processor combinations, then it is strongly recommended that the result of `defaultLookup()` exposes, as much as possible, a consistent set of symbols across all the OS and processor combinations.

Returns:

a symbol lookup for symbols in a set of commonly used libraries.

downcallType

`static MethodType downcallType(FunctionDescriptor`[PREVIEW] `functionDescriptor)`

Returns the downcall method handle type associated with the given function descriptor.

Parameters:

`functionDescriptor` - a function descriptor.

Returns:

the downcall method handle type associated with the given function descriptor

Throws:

`IllegalArgumentException` - if one or more layouts in the function descriptor are not supported (e.g. if they are sequence layouts or padding layouts).

upcallType

`static MethodType upcallType(FunctionDescriptor`[PREVIEW] `functionDescriptor)`

Returns the method handle type associated with an upcall stub with the given function descriptor.

Parameters:

`functionDescriptor` - a function descriptor.

Returns:

the method handle type associated with an upcall stub with the given function descriptor

Throws:

`IllegalArgumentException` - if one or more layouts in the function descriptor are not supported (e.g. if they are sequence layouts or padding layouts).

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.