Module jdk.incubator.foreign
Package jdk.incubator.foreign

Interface CLinker

All Superinterfaces:

SymbolLookup

```
public sealed interface CLinker
extends SymbolLookup
```

A C linker implements the C Application Binary Interface (ABI) calling conventions. Instances of this interface can be used to link foreign functions in native libraries that follow the JVM's target platform C ABI. A C linker provides two main capabilities: first, it allows Java code to *link* foreign functions into a so called *downcall method handle*; secondly, it allows native code to call Java method handles via the generation of *upcall stubs*.

On unsupported platforms this class will fail to initialize with an `ExceptionInInitializerError`.

Unless otherwise specified, passing a `null` argument, or an array argument containing one or more `null` elements to a method in this class causes a `NullPointerException` to be thrown.

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 native 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` 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`, 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 native symbol, the downcall method handle type features a leading parameter of type `NativeSymbol`, from which the address of the target native 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`, which is used by the linker runtime to allocate the memory region associated with the struct returned by the downcall method handle.

Variadic functions, declared in C either 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 native variadic function by using a *variadic* function descriptor, in which the specialized signature of a given variable arity callsite is described in full. Alternatively, if the foreign library allows it, clients might also be able to interact with variable arity methods by passing a trailing parameter of type `VaList`.

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` 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`, then `C` is set to `MemorySegment.class`

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

System lookup

This class implements the `SymbolLookup` interface; as such clients can look up symbols in the standard libraries associated with this linker. The set of symbols available for lookup is unspecified, as it depends on the platform and on the operating system.

Safety considerations

Obtaining downcall method handle is intrinsically unsafe. A symbol in a native library does not, in general, contain enough signature information (e.g. arity and types of native 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`, `NativeSymbol` or `VaList`):

- The resource scope of `R` is alive. Otherwise, the invocation throws `IllegalStateException`;
- The invocation occurs in same thread as the one `owning` the resource scope of `R`, if said scope is confined. Otherwise, the invocation throws `IllegalStateException`; and
- The scope of `R` is kept alive (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 native 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 native address, 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.

Method Summary

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

Modifier and Type	Method	Description
MethodHandle	downcallHandle (FunctionDescriptor functi	Obtains a foreign method handle, with the given type and featuring the given function descriptor, which can be used to call a target foreign function at the address in a dynamically provided native symbol.
default MethodHandle	downcallHandle (NativeSymbol symbol, FunctionDescriptor functio	Obtains a foreign method handle, with the given type and featuring the given function descriptor, which can be used to call a target foreign function at the address in the given native symbol.
static MethodType	downcallType (FunctionDescriptor functi	Obtains the downcall method handle type associated with a given function descriptor.
default Optional<NativeSymbol>	lookup(String name)	Look up a symbol in the standard libraries associated with this linker.
static CLinker	systemCLinker()	Returns the C linker for the current platform.
NativeSymbol	upcallStub (MethodHandle target, FunctionDescriptor functio ResourceScope scope)	Allocates a native stub with given scope which can be passed to other foreign functions (as a function pointer); calling such a function pointer from native code will result in the execution of the provided method handle

		method handle.
static MethodType	upcallType (FunctionDescriptor functi	Obtains the method handle type associated with an upcall stub with given function descriptor.

Method Details

systemCLinker

```
static CLinker systemCLinker()
```

Returns the C linker for the current platform.

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.

Returns:

a linker for this system.

Throws:

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

lookup

```
default Optional<NativeSymbol> lookup(String name)
```

Look up a symbol in the standard libraries associated with this linker. The set of symbols available for lookup is unspecified, as it depends on the platform and on the operating system.

Specified by:

`lookup` in interface `SymbolLookup`

Parameters:

`name` - the symbol name.

Returns:

a symbol in the standard libraries associated with this linker.

downcallHandle

```
default MethodHandle downcallHandle(NativeSymbol symbol,
                                    FunctionDescriptor function)
```

Obtains a foreign method handle, with the given type and featuring the given function descriptor, which can be used to call a target foreign function at the address in the given native symbol.

If the provided method type's return type is `MemorySegment`, then the resulting method handle features an additional prefix parameter, of type `SegmentAllocator`, 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` - downcall symbol.

`function` - the function descriptor.

Returns:

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

Throws:

`IllegalArgumentException` - if the provided descriptor contains either a sequence or a padding layout, or if the symbol is `MemoryAddress.NULL`

See Also:

`SymbolLookup`

downcallHandle

`MethodHandle downcallHandle(FunctionDescriptor function)`

Obtains a foreign method handle, with the given type and featuring the given function descriptor, which can be used to call a target foreign function at the address in a dynamically provided native symbol. The resulting method handle features a prefix parameter (as the first parameter) corresponding to the foreign function entry point, of type `NativeSymbol`.

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

The returned method handle will throw an `IllegalArgumentException` if the native symbol passed to it is associated with the `MemoryAddress.NULL` address, or a `NullPointerException` if the native symbol is `null`.

Parameters:

`function` - the function descriptor.

Returns:

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

Throws:

`IllegalArgumentException` - if the provided descriptor contains either a sequence or a padding layout.

See Also:

`SymbolLookup`

upcallStub

```
NativeSymbol upcallStub(MethodHandle target,
                        FunctionDescriptor function,
                        ResourceScope scope)
```

Allocates a native stub with given scope which can be passed to other foreign functions (as a function pointer); calling such a function pointer from native code will result in the execution of the provided method handle.

The returned function pointer is associated with the provided scope. When such scope is closed, the corresponding native 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 function descriptor.

`scope` - the upcall stub scope.

Returns:

the native stub symbol.

Throws:

`IllegalArgumentException` - if the provided descriptor contains either a sequence or a padding layout, or 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 `scope` has been already closed, or if access occurs from a thread other than the thread owning `scope`.

downcallType

```
static MethodType downcallType(FunctionDescriptor functionDescriptor)
```

Obtains the downcall method handle type associated with a given function descriptor.

Parameters:

`functionDescriptor` - a function descriptor.

Returns:

the downcall method handle type associated with a 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 functionDescriptor)
```

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

Parameters:

`functionDescriptor` - a function descriptor.

Returns:

the method handle type associated with an upcall stub with 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).
