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

Interface CLinker

`public interface CLinker`

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.

Linking a foreign function is a process which requires two components: a method type, and a function descriptor. The method type, consists of a set of *carrier* types, which, together, specify the Java signature which clients must adhere to when calling the underlying foreign function. The function descriptor contains a set of memory layouts which, together, specify the foreign function signature and classification information (via a custom layout attributes, see `CLinker.TypeKind`), so that linking can take place.

Clients of this API can build function descriptors using the predefined memory layout constants (based on a subset of the built-in types provided by the C language), found in this interface; alternatively, they can also decorate existing value layouts using the required `CLinker.TypeKind` classification attribute (this can be done using the `MemoryLayout.withAttribute(String, Constable)` method). A failure to do so might result in linkage errors, given that linking requires additional classification information to determine, for instance, how arguments should be loaded into registers during a foreign function call.

Implementations of this interface support the following primitive carrier types: `byte`, `short`, `char`, `int`, `long`, `float`, and `double`, as well as `MemoryAddress` for passing pointers, and `MemorySegment` for passing structs and unions. Finally, the `CLinker.VaList` carrier type can be used to match the native `va_list` type.

For the linking process to be successful, some requirements must be satisfied; if `M` and `F` are the method type and the function descriptor, respectively, used during the linking process, then it must be that:

- The arity of `M` is the same as that of `F`;
- If the return type of `M` is `void`, then `F` should have no return layout (see `FunctionDescriptor.ofVoid(MemoryLayout...)`);
- for each pair of carrier type `C` and layout `L` in `M` and `F`, respectively, where `C` and `L` refer to the same argument, or to the return value, the following conditions must hold:
  - If `C` is a primitve type, then `L` must be a `ValueLayout`, and the size of the layout must match that of the carrier type (see `Integer.SIZE` and similar fields in other primitive wrapper classes);
  - If `C` is `MemoryAddress.class`, then `L` must be a `ValueLayout`, and its size must match the platform's address size (see `MemoryLayouts.ADDRESS`). For this purpose, the `C_POINTER` layout constant can be used;
  - If `C` is `MemorySegment.class`, then `L` must be a `GroupLayout`
  - If `C` is `VaList.class`, then `L` must be `C_VA_LIST`

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. It is not

possible to create a method handle that takes a variable number of arguments, and neither is it possible to create an upcall stub wrapping a method handle that accepts a variable number of arguments. However, for downcalls only, it is possible to link a native variadic function by using a *specialized* method type and function descriptor: for each argument that is to be passed as a variadic argument, an explicit, additional, carrier type and memory layout must be present in the method type and function descriptor objects passed to the linker. Furthermore, as memory layouts corresponding to variadic arguments in a function descriptor must contain additional classification information, it is required that `asVarArg(MemoryLayout)` is used to create the memory layouts for each parameter corresponding to a variadic argument in a specialized function descriptor.

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.

API Note:

In the future, if the Java language permits, `CLinker` may become a `sealed` interface, which would prohibit subclassing except by explicitly permitted types.

Implementation Requirements:

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

Nested Class Summary

Nested Classes

Modifier and Type	Interface	Description
`static class`	**`CLinker.TypeKind`**	A C type kind.
`static interface`	**`CLinker.VaList`**	An interface that models a C `va_list`.

Field Summary

Fields

Modifier and Type	Field	Description
`static ValueLayout`	**`C_CHAR`**	The layout for the `char` C type
`static ValueLayout`	**`C_DOUBLE`**	The layout for the `double` C type
`static ValueLayout`	**`C_FLOAT`**	The layout for the `float` C type
`static ValueLayout`	**`C_INT`**	The layout for the `int` C type
`static ValueLayout`	**`C_LONG`**	The layout for the `long` C type
`static ValueLayout`	**`C_LONG_LONG`**	The layout for the `long long` C type.
`static ValueLayout`	**`C_POINTER`**	The `T*` native type.
`static ValueLayout`	**`C_SHORT`**	The layout for the `short` C type

static ValueLayout C_SHORT	The layout for the short C type
static MemoryLayout C_VA_LIST	The layout for the va_list C type

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
static MemoryAddress	allocateMemoryRestricted (long size)	Allocates memory of given size using malloc.
static <T extends MemoryLayout> T	asVarArg(T layout)	Returns a memory layout that is suitable to use as the layout for variadic arguments in a specialized function descriptor.
MethodHandle	downcallHandle (Addressable symbol, MethodType type, FunctionDescriptor functio	Obtain a foreign method handle, with given type, which can be used to call a target foreign function at a given address and featuring a given function descriptor.
static void	freeMemoryRestricted (MemoryAddress addr)	Frees the memory pointed by the given memory address.
static CLinker	getInstance()	Returns the C linker for the current platform.
static MemorySegment	toCString(String str)	Converts a Java string into a null-terminated C string, using the platform's default charset, storing the result into a new native memory segment.
static MemorySegment	toCString(String str, Charset charset)	Converts a Java string into a null-terminated C string, using the given charset, storing the result into a new native memory segment.
static MemorySegment	toCString(String str, Charset charset, NativeScope scope)	Converts a Java string into a null-terminated C string, using the given charset, storing the result into a new native memory segment native memory segment allocated using the provided scope.

static MemorySegment	toCString(String str, NativeScope scope)	Converts a Java string into a null-terminated C string, using the platform's default charset, storing the result into a native memory segment allocated using the provided scope.
static String	toJavaString (MemorySegment addr)	Converts a null-terminated C string stored at given address into a Java string, using the platform's default charset.
static String	toJavaString (MemorySegment addr, Charset charset)	Converts a null-terminated C string stored at given address into a Java string, using the given charset.
static String	toJavaStringRestricted (MemoryAddress addr)	Converts a null-terminated C string stored at given address into a Java string, using the platform's default charset.
static String	toJavaStringRestricted (MemoryAddress addr, Charset charset)	Converts a null-terminated C string stored at given address into a Java string, using the given charset.
MemorySegment	upcallStub (MethodHandle target, FunctionDescriptor functio	Allocates a native segment whose base address (see MemorySegment.address()) 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.

Field Details

C_CHAR

```
static final ValueLayout C_CHAR
```

The layout for the `char` C type

C_SHORT

`static final ValueLayout C_SHORT`

The layout for the `short` C type

C_INT

`static final ValueLayout C_INT`

The layout for the `int` C type

C_LONG

`static final ValueLayout C_LONG`

The layout for the `long` C type

C_LONG_LONG

`static final ValueLayout C_LONG_LONG`

The layout for the `long long` C type.

C_FLOAT

`static final ValueLayout C_FLOAT`

The layout for the `float` C type

C_DOUBLE

`static final ValueLayout C_DOUBLE`

The layout for the `double` C type

C_POINTER

`static final ValueLayout C_POINTER`

The `T*` native type.

C_VA_LIST

`static final MemoryLayout C_VA_LIST`

The layout for the `va_list` C type

Method Details

getInstance

```
static CLinker getInstance()
```

Returns the C linker for the current platform.

This method is *restricted*. Restricted method 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:

`IllegalAccessError` - if the runtime property `foreign.restricted` is not set to either `permit`, `warn` or `debug` (the default value is set to `deny`).

downcallHandle

```
MethodHandle downcallHandle(Addressable symbol,
                            MethodType type,
                            FunctionDescriptor function)
```

Obtain a foreign method handle, with given type, which can be used to call a target foreign function at a given address and featuring a given function descriptor.

Parameters:

`symbol` - downcall symbol.

`type` - the method type.

`function` - the function descriptor.

Returns:

the downcall method handle.

Throws:

`IllegalArgumentException` - in the case of a method type and function descriptor mismatch.

See Also:

`LibraryLookup.lookup(String)`

upcallStub

```
MemorySegment upcallStub(MethodHandle target,
                         FunctionDescriptor function)
```

Allocates a native segment whose base address (see `MemorySegment.address()`) 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 segment is shared, and it only features the `MemorySegment.CLOSE` access mode. When the returned segment is closed, the corresponding native stub will be deallocated.

Parameters:

`target` - the target method handle.

`function` - the function descriptor.

Returns:

the native stub segment.

Throws:

`IllegalArgumentException` - if the target's method type and the function descriptor mismatch.

asVarArg

```
static <T extends MemoryLayout> T asVarArg(T layout)
```

Returns a memory layout that is suitable to use as the layout for variadic arguments in a specialized function descriptor.

Type Parameters:

`T` - the memory layout type

Parameters:

`layout` - the layout the adapt

Returns:

a potentially newly created layout with the right attributes

toCString

```
static MemorySegment toCString(String str)
```

Converts a Java string into a null-terminated C string, using the platform's default charset, storing the result into a new native memory segment.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array. The `CharsetEncoder` class should be used when more control over the encoding process is required.

Parameters:

`str` - the Java string to be converted into a C string.

Returns:

a new native memory segment containing the converted C string.

toCString

```
static MemorySegment toCString(String str,
                               Charset charset)
```

Converts a Java string into a null-terminated C string, using the given `charset`, storing the result into a new native memory segment.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array. The `CharsetEncoder` class should be used when more control over the encoding process is required.

Parameters:

`str` - the Java string to be converted into a C string.

`charset` - The `Charset` to be used to compute the contents of the C string.

Returns:

a new native memory segment containing the converted C string.

toCString

```
static MemorySegment toCString(String str,
                               NativeScope scope)
```

Converts a Java string into a null-terminated C string, using the platform's default charset, storing the result into a native memory segment allocated using the provided scope.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array. The `CharsetEncoder` class should be used when more control over the encoding process is required.

Parameters:

`str` - the Java string to be converted into a C string.

`scope` - the scope to be used for the native segment allocation.

Returns:

a new native memory segment containing the converted C string.

toCString

```
static MemorySegment toCString(String str,
                               Charset charset,
                               NativeScope scope)
```

Converts a Java string into a null-terminated C string, using the given `charset`, storing the result into a new native memory segment native memory segment allocated using the provided scope.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array. The `CharsetEncoder` class should be used when more control over the encoding process is required.

Parameters:

`str` - the Java string to be converted into a C string.

`charset` - The `Charset` to be used to compute the contents of the C string.

`scope` - the scope to be used for the native segment allocation.

Returns:

a new native memory segment containing the converted C string.

toJavaStringRestricted

```
static String toJavaStringRestricted(MemoryAddress addr)
```

Converts a null-terminated C string stored at given address into a Java string, using the platform's default charset.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

This method is *restricted*. Restricted method 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.

Parameters:

`addr` - the address at which the string is stored.

Returns:

a Java string with the contents of the null-terminated C string at given address.

Throws:

`IllegalArgumentException` - if the size of the native string is greater than the largest string supported by the platform.

toJavaStringRestricted

```
static String toJavaStringRestricted(MemoryAddress addr,
                                     Charset charset)
```

Converts a null-terminated C string stored at given address into a Java string, using the given `charset`.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

This method is *restricted*. Restricted method 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.

Parameters:

`addr` - the address at which the string is stored.

`charset` - The `Charset` to be used to compute the contents of the Java string.

Returns:

a Java string with the contents of the null-terminated C string at given address.

Throws:

`IllegalArgumentException` - if the size of the native string is greater than the largest string supported by the platform.

toJavaString

```
static String toJavaString(MemorySegment addr)
```

Converts a null-terminated C string stored at given address into a Java string, using the platform's default charset.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`addr` - the address at which the string is stored.

Returns:

a Java string with the contents of the null-terminated C string at given address.

Throws:

`IllegalArgumentException` - if the size of the native string is greater than the largest string supported by the platform.

`IllegalStateException` - if the size of the native string is greater than the size of the segment associated with `addr`, or if `addr` is associated with a segment that is *not alive*.

toJavaString

```
static String toJavaString(MemorySegment addr,
                           Charset charset)
```

Converts a null-terminated C string stored at given address into a Java string, using the given `charset`.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`addr` - the address at which the string is stored.

`charset` - The `Charset` to be used to compute the contents of the Java string.

Returns:

a Java string with the contents of the null-terminated C string at given address.

Throws:

`IllegalArgumentException` - if the size of the native string is greater than the largest string supported by the platform.

`IllegalStateException` - if the size of the native string is greater than the size of the segment associated with `addr`, or if `addr` is associated with a segment that is *not alive*.

allocateMemoryRestricted

```
static MemoryAddress allocateMemoryRestricted(long size)
```

Allocates memory of given size using malloc.

This method is *restricted*. Restricted method 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.

Parameters:

`size` - memory size to be allocated

Returns:

addr memory address of the allocated memory

Throws:

`OutOfMemoryError` - if malloc could not allocate the required amount of native memory.

freeMemoryRestricted

```
static void freeMemoryRestricted(MemoryAddress addr)
```

Frees the memory pointed by the given memory address.

This method is *restricted*. Restricted method 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.

Parameters:

`addr` - memory address of the native memory to be freed