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

Interface ResourceScope

All Superinterfaces:

AutoCloseable

```
public sealed interface ResourceScope
extends AutoCloseable
```

A resource scope manages the lifecycle of one or more resources. Resources (e.g. `MemorySegment`) associated with a resource scope can only be accessed while the resource scope is alive, and by the thread associated with the resource scope (if any).

Deterministic deallocation

Resource scopes support *deterministic deallocation*; that is, they can be closed explicitly. When a resource scope is closed, it is no longer `alive`, and subsequent operations on resources associated with that scope (e.g. attempting to access a `MemorySegment` instance) will fail with `IllegalStateException`.

Closing a resource scope will cause all the close actions associated with that scope to be called. Moreover, closing a resource scope might trigger the releasing of the underlying memory resources associated with said scope; for instance:

- closing the scope associated with a native memory segment results in *freeing* the native memory associated with it;
- closing the scope associated with a mapped memory segment results in the backing memory-mapped file to be unmapped;
- closing the scope associated with an upcall stub results in releasing the stub;
- closing the scope associated with a variable arity list results in releasing the memory associated with that variable arity list instance.

Implicit deallocation

Resource scopes can be associated with a `Cleaner` instance, so that they are also closed automatically, once the scope instance becomes unreachable. This can be useful to allow for predictable, deterministic resource deallocation, while still preventing accidental native memory leaks. In case a managed resource scope is closed explicitly, no further action will be taken when the scope becomes unreachable; that is, close actions associated with a resource scope, whether managed or not, are called *exactly once*.

Global scope

An important implicit resource scope is the so called global scope; the global scope is a resource scope that cannot be closed, either explicitly or implicitly. As a result, the global scope will never attempt to release resources associated with it. Examples of resources associated with the global scope are:

- heap segments created from arrays or buffers;
- variable arity lists obtained from raw memory addresses;
- native symbols obtained from a loader lookup, or from the `CLinker`.

In other words, the global scope is used to indicate that the lifecycle of one or more resources must, where needed, be managed independently by clients.

Thread confinement

Resource scopes can be divided into two categories: *thread-confined* resource scopes, and *shared* resource scopes.

Confined resource scopes, support strong thread-confinement guarantees. Upon creation, they are assigned an owner thread, typically the thread which initiated the creation operation. After creating a confined resource scope, only the owner thread will be allowed to directly manipulate the resources associated with this resource scope. Any attempt to perform resource access from a thread other than the owner thread will result in a runtime failure.

Shared resource scopes, on the other hand, have no owner thread; as such, resources associated with shared resource scopes can be accessed by multiple threads. This might be useful when multiple threads need to access the same resource concurrently (e.g. in the case of parallel processing). For instance, a client might obtain a `Spliterator` from a segment backed by a shared scope, which can then be used to slice the segment and allow multiple threads to work in parallel on disjoint segment slices. The following code can be used to sum all int values in a memory segment in parallel:

```
try (ResourceScope scope = ResourceScope.newSharedScope()) {
    SequenceLayout SEQUENCE_LAYOUT = MemoryLayout.sequenceLayout(1024, ValueLayout.JAVA_INT);
    MemorySegment segment = MemorySegment.allocateNative(SEQUENCE_LAYOUT, scope);
    int sum = segment.elements(ValueLayout.JAVA_INT).parallel()
                     .mapToInt(s -> s.get(ValueLayout.JAVA_INT, 0))
                     .sum();
}
```

Shared resource scopes, while powerful, must be used with caution: if one or more threads accesses a resource associated with a shared scope while the scope is being closed from another thread, an exception might occur on both the accessing and the closing threads. Clients should refrain from attempting to close a shared resource scope repeatedly (e.g. keep calling `close()` until no exception is thrown). Instead, clients of shared resource scopes should always ensure that proper synchronization mechanisms (e.g. using temporal dependencies, see below) are put in place so that threads closing shared resource scopes can never race against threads accessing resources managed by same scopes.

Temporal dependencies

Resource scopes can depend on each other. More specifically, a scope can feature temporal dependencies on one or more other resource scopes. Such a resource scope cannot be closed (either implicitly or explicitly) until *all* the scopes it depends on have also been closed.

This can be useful when clients need to perform a critical operation on a memory segment, during which they have to ensure that the scope associated with that segment will not be closed; this can be done as follows:

```
MemorySegment segment = ...
try (ResourceScope criticalScope = ResourceScope.newConfinedScope()) {
    criticalScope.keepAlive(segment.scope());
    <critical operation on segment>
}
```

Note that a resource scope does not become unreachable until all the scopes it depends on have been closed.

Implementation Requirements:

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

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
void	**addCloseAction**(**Runnable** runnable)	Add a custom cleanup action which will be executed when the resource scope is closed.
void	**close**()	Closes this resource scope.
static **ResourceScope**	**globalScope**()	Returns the *global scope*.
boolean	**isAlive**()	Returns `true`, if this resource scope is alive.
void	**keepAlive**(**ResourceScope** target)	Creates a temporal dependency between this scope and the target scope.
static **ResourceScope**	**newConfinedScope**()	Creates a new confined scope.
static **ResourceScope**	**newConfinedScope**(**Cleaner** cleaner)	Creates a new confined scope, managed by the provided cleaner instance.
static **ResourceScope**	**newImplicitScope**()	Creates a new shared scope, managed by a private `Cleaner` instance.
static **ResourceScope**	**newSharedScope**()	Creates a new shared scope.
static **ResourceScope**	**newSharedScope**(**Cleaner** cleaner)	Creates a new shared scope, managed by the provided cleaner instance.
Thread	**ownerThread**()	The thread owning this resource scope.

Method Details

isAlive

```
boolean isAlive()
```

Returns `true`, if this resource scope is alive.

Returns:

`true`, if this resource scope is alive

See Also:

`close()`

ownerThread

```
Thread ownerThread()
```

The thread owning this resource scope.

Returns:

the thread owning this resource scope, or `null` if this resource scope is shared.

close

`void close()`

Closes this resource scope. As a side effect, if this operation completes without exceptions, this scope will be marked as *not alive*, and subsequent operations on resources associated with this scope will fail with `IllegalStateException`. Additionally, upon successful closure, all native resources associated with this resource scope will be released.

Specified by:

`close` in interface `AutoCloseable`

API Note:

This operation is not idempotent; that is, closing an already closed resource scope *always* results in an exception being thrown. This reflects a deliberate design choice: resource scope state transitions should be manifest in the client code; a failure in any of these transitions reveals a bug in the underlying application logic.

Throws:

`IllegalStateException` - if one of the following condition is met:

- this resource scope is not *alive*
- this resource scope is confined, and this method is called from a thread other than the thread owning this resource scope
- this resource scope is shared and a resource associated with this scope is accessed while this method is called
- one or more scopes which depend on this resource scope have not been closed.

`UnsupportedOperationException` - if this resource scope is the global scope.

addCloseAction

`void addCloseAction(Runnable runnable)`

Add a custom cleanup action which will be executed when the resource scope is closed. The order in which custom cleanup actions are invoked once the scope is closed is unspecified.

Parameters:

`runnable` - the custom cleanup action to be associated with this scope.

Throws:

`IllegalStateException` - if this scope has been closed, or if access occurs from a thread other than the thread owning this scope.

keepAlive

`void keepAlive(ResourceScope target)`

Creates a temporal dependency between this scope and the target scope. As a result, the target scope cannot be closed *before* this scope.

Implementation Note:

A given scope can support up to `Integer.MAX_VALUE` pending keep alive requests.

Parameters:

`target` - the scope that needs to be kept alive.

Throws:

`IllegalArgumentException` - if `target == this`.

`IllegalStateException` - if this scope or `target` have been closed, or if access occurs from a thread other than the thread owning this scope or `target`.

newConfinedScope

`static ResourceScope newConfinedScope()`

Creates a new confined scope.

Returns:

a new confined scope.

newConfinedScope

`static ResourceScope newConfinedScope(Cleaner cleaner)`

Creates a new confined scope, managed by the provided cleaner instance.

Parameters:

`cleaner` - the cleaner to be associated with the returned scope.

Returns:

a new confined scope, managed by `cleaner`.

newSharedScope

```
static ResourceScope newSharedScope()
```

Creates a new shared scope.

Returns:
a new shared scope.

newSharedScope

```
static ResourceScope newSharedScope(Cleaner cleaner)
```

Creates a new shared scope, managed by the provided cleaner instance.

Parameters:
`cleaner` - the cleaner to be associated with the returned scope.

Returns:
a new shared scope, managed by `cleaner`.

newImplicitScope

```
static ResourceScope newImplicitScope()
```

Creates a new shared scope, managed by a private `Cleaner` instance. Equivalent to (but likely more efficient than) the following code:

```
newSharedScope(Cleaner.create());
```

Returns:
a shared scope, managed by a private `Cleaner` instance.

globalScope

```
static ResourceScope globalScope()
```

Returns the *global scope*.

Returns:
the *global scope*.