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* (see isAlive()), and by the thread associated with the resource scope (if any).

Explicit resource scopes

Resource scopes obtained from newConfinedScope(), newSharedScope() support *deterministic deallocation*; We call these resource scopes *explicit scopes*. Explicit resource scopes can be closed explicitly (see close()). When a resource scope is closed, it is no longer *alive* (see isAlive(), 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 cleanup actions associated with that scope (see addCloseAction(Runnable)) 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 (see MemorySegment.allocateNative(long, ResourceScope), or SegmentAllocator.arenaAllocator(ResourceScope))
- closing the scope associated with a mapped memory segment results in the backing memory-mapped file to be unmapped (see MemorySegment.mapFile(Path, long, long, FileChannel.MapMode, ResourceScope))
- closing the scope associated with an upcall stub results in releasing the stub (see CLinker.upcallStub(MethodHandle, FunctionDescriptor, ResourceScope)

Sometimes, explicit scopes can be associated with a Cleaner instance (see newConfinedScope(Cleaner) and newSharedScope(Cleaner)). We call these resource scopes *managed* resource scopes. A managed resource scope is closed automatically once the scope instance becomes unreachable.

Managed scopes can be useful to allow for predictable, deterministic resource deallocation, while still prevent 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, cleanup actions (see addCloseAction(Runnable)) associated with a resource scope, whether managed or not, are called *exactly once*.

Implicit resource scopes

Resource scopes obtained from newImplicitScope() cannot be closed explicitly. We call these resource scopes *implicit scopes*. Calling close() on an implicit resource scope always results in an exception. Resources associated with implicit scopes are released once the scope instance becomes unreachable.

An important implicit resource scope is the so called global scope; the global scope is an implicit scope that is guaranteed to never become unreachable. As a results, the global scope will never attempt to release resources associated with it. Such resources must, where needed, be managed independently by clients.

Thread confinement

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

Confined resource scopes (see newConfinedScope()), support strong thread-confinement guarantees. Upon creation, they are assigned an *owner thread*, typically the thread which initiated the creation operation (see ownerThread()). 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 (see newSharedScope() and newImplicitScope()), on the other hand, have no owner thread; as such resources associated with this 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 shared segment, 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:

```
SequenceLayout SEQUENCE_LAYOUT = MemoryLayout.sequenceLayout(1024, MemoryLayouts.JAVA_INT);
try (ResourceScope scope = ResourceScope.newSharedScope()) {
    MemorySegment segment = MemorySegment.allocateNative(SEQUENCE_LAYOUT, scope);
    VarHandle VH_int = SEQUENCE_LAYOUT.elementLayout().varHandle(int.class);
    int sum = StreamSupport.stream(segment.spliterator(SEQUENCE_LAYOUT), true)
        .mapToInt(s -> (int)VH_int.get(s.address()))
        .sum();
}
```

Explicit 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 resource scope handles, see below) are put in place so that threads closing shared resource scopes can never race against threads accessing resources managed by same scopes.

Resource scope handles

Resource scopes can be made *non-closeable* by acquiring one or more resource scope *handles* (see `acquire()`. A resource scope handle can be used to make sure that resources associated with a given resource scope (either explicit or implicit) cannot be released for a certain period of time - e.g. during a critical region of code involving one or more resources associated with the scope. For instance, an explicit resource scope can only be closed *after* all the handles acquired against that scope have been closed (see `close()`). This can be useful when clients need to perform a critical operation on a memory segment, during which they have to ensure that the segment will not be released; this can be done as follows:

```
MemorySegment segment = ...
ResourceScope.Handle segmentHandle = segment.scope().acquire()
try {
   <critical operation on segment>
} finally {
   segment.scope().release(segmentHandle);
}
```

Acquiring implicit resource scopes is also possible, but it is often unnecessary: since resources associated with an implicit scope will only be released when the scope becomes unreachable, clients can use e.g. `Reference.reachabilityFence(Object)` to make sure that resources associated with implicit scopes are not released prematurely. That said, the above code snippet works (trivially) for implicit scopes too.

Implementation Requirements:

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

Nested Class Summary

Nested Classes

Modifier and Type	Interface	Description
`static interface`	`ResourceScope.Handle`	An abstraction modelling a resource scope handle.

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method	Description
`ResourceScope.Handle`	`acquire()`	Acquires a resource scope handle associated with this resource scope.
`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 an implicit scope which is assumed to be always alive.
`boolean`	`isAlive()`	Is this resource scope alive?
`boolean`	`isImplicit()`	Is this resource scope an *implicit scope*?
`static ResourceScope`	`newConfinedScope()`	Create a new confined scope.
`static ResourceScope`	`newConfinedScope(Cleaner cleaner)`	Create a new confined scope managed by a `Cleaner`.
`static ResourceScope`	`newImplicitScope()`	Create a new *implicit scope*.
`static ResourceScope`	`newSharedScope()`	Create a new shared scope.
`static ResourceScope`	`newSharedScope(Cleaner cleaner)`	Create a new shared scope managed by a `Cleaner`.
`Thread`	`ownerThread()`	The thread owning this resource scope.
`void`	`release(ResourceScope.Handle handle)`	Release the provided resource scope handle.

Method Details

isAlive

```
boolean isAlive()
```

Is this resource scope 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.

isImplicit

```
boolean isImplicit()
```

Is this resource scope an *implicit scope*?

Returns:

true if this scope is an *implicit scope*.

See Also:

`newImplicitScope()`, `globalScope()`

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 handles (see `acquire()`) associated with this resource scope have not been released

`UnsupportedOperationException` - if this resource scope is implicit.

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 already been closed.

acquire

```
ResourceScope.Handle acquire()
```

Acquires a resource scope handle associated with this resource scope. An explicit resource scope cannot be closed until all the resource scope handles acquired from it have been release(Handle) released}.

Returns:

a resource scope handle.

release

`void release(ResourceScope.Handle handle)`

Release the provided resource scope handle. This method is idempotent, that is, releasing the same handle multiple times has no effect.

Parameters:

`handle` - the resource scope handle to be released.

Throws:

`IllegalArgumentException` - if the provided handle is not associated with this scope.

newConfinedScope

`static ResourceScope newConfinedScope()`

Create a new confined scope. The resulting scope is closeable, and is not managed by a `Cleaner`.

Returns:

a new confined scope.

newConfinedScope

`static ResourceScope newConfinedScope(Cleaner cleaner)`

Create a new confined scope managed by a `Cleaner`.

Parameters:

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

Returns:

a new confined scope, managed by `cleaner`.

Throws:

`NullPointerException` - if `cleaner == null`.

newSharedScope

`static ResourceScope newSharedScope()`

Create a new shared scope. The resulting scope is closeable, and is not managed by a `Cleaner`.

Returns:

a new shared scope.

newSharedScope

`static ResourceScope newSharedScope(Cleaner cleaner)`

Create a new shared scope managed by a `Cleaner`.

Parameters:

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

Returns:

a new shared scope, managed by `cleaner`.

Throws:

`NullPointerException` - if `cleaner == null`.

newImplicitScope

`static ResourceScope newImplicitScope()`

Create a new *implicit scope*. The implicit scope is a managed, shared, and non-closeable scope which only features *implicit closure*. Since implicit scopes can only be closed implicitly by the garbage collector, it is recommended that implicit scopes are only used in cases where deallocation performance is not a critical concern, to avoid unnecessary memory pressure.

Returns:

a new implicit scope.

globalScope

```
static ResourceScope globalScope()
```

Returns an implicit scope which is assumed to be always alive.

Returns:

the global scope.
