

Module `jdk.incubator.concurrent`

Package `jdk.incubator.concurrent`

Class `StructuredTaskScope<T>`

`java.lang.Object`

`jdk.incubator.concurrent.StructuredTaskScope<T>`

Type Parameters:

T - the result type of tasks executed in the scope

All Implemented Interfaces:

`AutoCloseable`

Direct Known Subclasses:

`StructuredTaskScope.ShutdownOnFailure`, `StructuredTaskScope.ShutdownOnSuccess`

```
public class StructuredTaskScope<T>
    extends Object
    implements AutoCloseable
```

A basic API for *structured concurrency*. `StructuredTaskScope` supports cases where a task splits into several concurrent subtasks, to be executed in their own threads, and where the subtasks must complete before the main task continues. A `StructuredTaskScope` can be used to ensure that the lifetime of a concurrent operation is confined by a *syntax block*, just like that of a sequential operation in structured programming.

Basic usage

A `StructuredTaskScope` is created with one of its public constructors. It defines the `fork` method to start a thread to execute a task, the `join` method to wait for all threads to finish, and the `close` method to close the task scope. The API is intended to be used with the try-with-resources construct. The intention is that code in the *block* uses the `fork` method to fork threads to execute the subtasks, wait for the threads to finish with the `join` method, and then *process the results*. Processing of results may include handling or re-throwing of exceptions.

```
try (var scope = new StructuredTaskScope<Object>()) {

    Future<Integer> future1 = scope.fork(task1);
    Future<String> future2 = scope.fork(task2);

    scope.join();

    ... process results/exceptions ...

} // close
```

To ensure correct usage, the `join` and `close` methods may only be invoked by the *owner* (the thread that opened/created the task scope), and the `close` method throws an exception after closing if the owner did not invoke the `join` method after forking.

StructuredTaskScope defines the `shutdown` method to shut down a task scope without closing it. Shutdown is useful for cases where a subtask completes with a result (or exception) and the results of other unfinished subtasks are no longer needed. If a subtask invokes shutdown while the owner is waiting in the `join` method then it will cause `join` to wakeup, all unfinished threads to be `interrupted` and prevents new threads from starting in the task scope.

Subclasses with policies for common cases

Two subclasses of StructuredTaskScope are defined to implement policy for common cases:

1. `ShutdownOnSuccess` captures the first result and shuts down the task scope to interrupt unfinished threads and wakeup the owner. This class is intended for cases where the result of any subtask will do ("invoke any") and where there is no need to wait for results of other unfinished tasks. It defines methods to get the first result or throw an exception if all subtasks fail.
2. `ShutdownOnFailure` captures the first exception and shuts down the task scope. This class is intended for cases where the results of all subtasks are required ("invoke all"); if any subtask fails then the results of other unfinished subtasks are no longer needed. It defines methods to throw an exception if any of the subtasks fail.

The following are two examples that use the two classes. In both cases, a pair of subtasks are forked to fetch resources from two URL locations "left" and "right". The first example creates a `ShutdownOnSuccess` object to capture the result of the first subtask to complete normally, cancelling the other by way of shutting down the task scope. The main task waits in `join` until either subtask completes with a result or both subtasks fail. It invokes `result(Function)` method to get the captured result. If both subtasks fail then this method throws a `WebApplicationException` with the exception from one of the subtasks as the cause.

```
try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {  
  
    scope.fork(() -> fetch(left));  
    scope.fork(() -> fetch(right));  
  
    scope.join();  
  
    String result = scope.result(e -> new WebApplicationException(e));  
  
    ...  
}
```

The second example creates a `ShutdownOnFailure` object to capture the exception of the first subtask to fail, cancelling the other by way of shutting down the task scope. The main task waits in `joinUntil(Instant)` until both subtasks complete with a result, either fails, or a deadline is reached. It invokes `throwIfFailed(Function)` to throw an exception when either subtask fails. This method is a no-op if no subtasks fail. The main task uses `Future's resultNow()` method to retrieve the results.

```
Instant deadline = ...  
  
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
```

```

Future<String> future1 = scope.fork(() -> query(left));
Future<String> future2 = scope.fork(() -> query(right));

scope.joinUntil(deadline);

scope.throwIfFailed(e -> new WebApplicationException(e));

// both subtasks completed successfully
String result = Stream.of(future1, future2)
    .map(Future::resultNow)
    .collect(Collectors.joining(", ", "{ ", " }"));

...
}

```

Extending StructuredTaskScope

StructuredTaskScope can be extended, and the `handleComplete` overridden, to implement policies other than those implemented by `ShutdownOnSuccess` and `ShutdownOnFailure`. The method may be overridden to, for example, collect the results of subtasks that complete with a result and ignore subtasks that fail. It may collect exceptions when subtasks fail. It may invoke the `shutdown` method to shut down and cause `join` to wakeup when some condition arises.

A subclass will typically define methods to make available results, state, or other outcome to code that executes after the `join` method. A subclass that collects results and ignores subtasks that fail may define a method that returns a collection of results. A subclass that implements a policy to shut down when a subtask fails may define a method to retrieve the exception of the first subtask to fail.

The following is an example of a `StructuredTaskScope` implementation that collects the results of subtasks that complete successfully. It defines the method `results()` to be used by the main task to retrieve the results.

```


class MyScope<T> extends StructuredTaskScope<T> {
    private final Queue<T> results = new ConcurrentLinkedQueue<>();

    MyScope() {
        super(null, Thread.ofVirtual().factory());
    }

    @Override
    protected void handleComplete(Future<T> future) {
        if (future.state() == Future.State.SUCCESS) {
            T result = future.resultNow();
            results.add(result);
        }
    }

    // Returns a stream of results from the subtasks that completed successfully
    public Stream<T> results() {
        return results.stream();
    }
}

```

Copy 

```
}
```

Tree structure

Task scopes form a tree where parent-child relations are established implicitly when opening a new task scope:

- A parent-child relation is established when a thread started in a task scope opens its own task scope. A thread started in task scope "A" that opens task scope "B" establishes a parent-child relation where task scope "A" is the parent of task scope "B".
- A parent-child relation is established with nesting. If a thread opens task scope "B", then opens task scope "C" (before it closes "B"), then the enclosing task scope "B" is the parent of the nested task scope "C".

The *descendants* of a task scope are the child task scopes that it is a parent of, plus the descendants of the child task scopes, recursively.

The tree structure supports:

- Inheritance of [scoped values](#) across threads.
- Confinement checks. The phrase "threads contained in the task scope" in method

The following example demonstrates the inheritance of a scoped value. A scoped value `USERNAME` is bound to the value "duke". A `StructuredTaskScope` is created and its `fork` method invoked to start a thread to execute `childTask`. The thread inherits the scoped value *bindings* captured when creating the task scope. The code in `childTask` uses the value of the scoped value and so reads the value "duke".

```
private static final ScopedValue<String> USERNAME = ScopedValue.newInstance();

ScopedValue.where(USERNAME, "duke", () -> {
    try (var scope = new StructuredTaskScope<String>()) {

        scope.fork(() -> childTask());
        ...
    }
});

...

String childTask() {
    String name = USERNAME.get();    // "duke"
    ...
}
```

`StructuredTaskScope` does not define APIs that exposes the tree structure at this time.

Unless otherwise specified, passing a `null` argument to a constructor or method in this class will cause a [NullPointerException](#) to be thrown.

Memory consistency effects

Actions in the owner thread of `of`, or a thread contained in, the task scope prior to `forking` of a Callable task *happen-before* any actions taken by that task, which in turn *happen-before* the task result is retrieved via its Future, or *happen-before* any actions taken in a thread after `joining` of the task scope.

See *Java Language Specification*:

17.4.5 Happens-before Order[↗]

Since:

19

Nested Class Summary

Nested Classes		
Modifier and Type	Class	Description
static final class	<code>StructuredTaskScope.ShutdownAwaiter</code>	A <code>StructuredTaskScope</code> that captures the exception of the first subtask to complete abnormally.
static final class	<code>StructuredTaskScope.ShutdownAwaitResult</code>	A <code>StructuredTaskScope</code> that captures the result of the first subtask to complete successfully.

Constructor Summary

Constructors	
Constructor	Description
<code>StructuredTaskScope()</code>	Creates an unnamed structured task scope that creates virtual threads.
<code>StructuredTaskScope(String name, ThreadFactory factory)</code>	Creates a structured task scope with the given name and thread factory.

Method Summary

All Methods		
Instance Methods		
Concrete Methods		
Modifier and Type	Method	Description
void	<code>close()</code>	Closes this task scope.
<U extends T> Future<U>	<code>fork(Callable<? extends U> task)</code>	Starts a new thread to run the given task.

protected void	handleComplete (Future < T > future)	Invoked when a task completes before the scope is shut down.
StructuredTaskScope < T > join ()		Wait for all threads to finish or the task scope to shut down.
StructuredTaskScope < T > joinUntil (Instant deadline)		Wait for all threads to finish or the task scope to shut down, up to the given deadline.
void	shutdown ()	Shut down the task scope without closing it.

Methods declared in class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Details

StructuredTaskScope

```
public StructuredTaskScope(String name,
                           ThreadFactory factory)
```

Creates a structured task scope with the given name and thread factory. The task scope is optionally named for the purposes of monitoring and management. The thread factory is used to [create](#) threads when tasks are [forked](#). The task scope is owned by the current thread.

This method captures the current thread's [scoped value](#) bindings for inheritance by threads created in the task scope. The [Tree Structure](#) section in the class description details how parent-child relations are established implicitly for the purpose of inheritance of scoped value bindings.

Parameters:

name - the name of the task scope, can be null

factory - the thread factory

StructuredTaskScope

```
public StructuredTaskScope()
```

Creates an unnamed structured task scope that creates virtual threads. The task scope is owned by the current thread.

This constructor is equivalent to invoking the 2-arg constructor with a name of `null` and a thread factory that creates virtual threads.

Throws:

`UnsupportedOperationException` - if preview features are not enabled

Method Details

handleComplete

```
protected void handleComplete(Future<T> future)
```

Invoked when a task completes before the scope is shut down.

The `handleComplete` method should be thread safe. It may be invoked by several threads concurrently.

Implementation Requirements:

The default implementation does nothing.

Parameters:

`future` - the completed task

fork

```
public <U extends T> Future<U> fork(Callable<? extends U> task)
```

Starts a new thread to run the given task.

The new thread is created with the task scope's `ThreadFactory`. It inherits the current thread's `scoped value` bindings. The bindings must match the bindings captured when the task scope was created.

If the task completes before the task scope is `shutdown` then the `handleComplete` method is invoked to consume the completed task. The `handleComplete` method is run when the task completes with a result or exception. If the `Future`'s `cancel` method is used to cancel a task before the task scope is shut down, then the `handleComplete` method is run by the thread that invokes `cancel`. If the task scope shuts down at or around the same time that the task completes or is cancelled then the `handleComplete` method may or may not be invoked.

If this task scope is `shutdown` (or in the process of shutting down) then `fork` returns a `Future` representing a `cancelled` task that was not run.

This method may only be invoked by the task scope owner or threads contained in the task scope. The `cancel` method of the returned `Future` object is also restricted to the task scope owner or threads contained in the task scope. The `cancel` method throws `WrongThreadException` if invoked from another thread. All other methods on the returned `Future` object, such as `get`, are not restricted.

Type Parameters:

U - the result type

Parameters:

task - the task to run

Returns:

a future

Throws:

[IllegalStateException](#) - if this task scope is closed

[WrongThreadException](#) - if the current thread is not the owner or a thread contained in the task scope

[StructureViolationException](#) - if the current scoped value bindings are not the same as when the task scope was created

[RejectedExecutionException](#) - if the thread factory rejected creating a thread to run the task

join

```
public StructuredTaskScope<T> join()  
    throws InterruptedException
```

Wait for all threads to finish or the task scope to shut down. This method waits until all threads started in the task scope finish execution (of both task and [handleComplete](#) method), the [shutdown](#) method is invoked to shut down the task scope, or the current thread is [interrupted](#).

This method may only be invoked by the task scope owner.

Returns:

this task scope

Throws:

[IllegalStateException](#) - if this task scope is closed

[WrongThreadException](#) - if the current thread is not the owner

[InterruptedException](#) - if interrupted while waiting

joinUntil

```
public StructuredTaskScope<T> joinUntil(Instant deadline)  
    throws InterruptedException,  
        TimeoutException
```

Wait for all threads to finish or the task scope to shut down, up to the given deadline. This method waits until all threads started in the task scope finish execution (of both task and [handleComplete](#) method), the [shutdown](#) method is invoked to shut down the task scope, the current thread is [interrupted](#), or the deadline is reached.

This method may only be invoked by the task scope owner.

Parameters:

deadline - the deadline

Returns:

this task scope

Throws:

[IllegalStateException](#) - if this task scope is closed

[WrongThreadException](#) - if the current thread is not the owner

[InterruptedException](#) - if interrupted while waiting

[TimeoutException](#) - if the deadline is reached while waiting

shutdown

```
public void shutdown()
```

Shut down the task scope without closing it. Shutting down a task scope prevents new threads from starting, interrupts all unfinished threads, and causes the [join](#) method to wakeup. Shutdown is useful for cases where the results of unfinished subtasks are no longer needed.

More specifically, this method:

- [Cancels](#) the tasks that have threads [waiting](#) on a result so that the waiting threads wakeup.
- [Interrupts](#) all unfinished threads in the task scope (except the current thread).
- Wakes up the owner if it is waiting in [join\(\)](#) or [joinUntil\(Instant\)](#). If the owner is not waiting then its next call to [join](#) or [joinUntil](#) will return immediately.

When this method completes then the Future objects for all tasks will be [done](#), normally or abnormally. There may still be threads that have not finished because they are executing code that did not respond (or respond promptly) to thread interrupt. This method does not wait for these threads. When the owner invokes the [close](#) method to close the task scope then it will wait for the remaining threads to finish.

This method may only be invoked by the task scope owner or threads contained in the task scope.

Throws:

[IllegalStateException](#) - if this task scope is closed

[WrongThreadException](#) - if the current thread is not the owner or a thread contained in the task scope

close

```
public void close()
```

Closes this task scope.

This method first shuts down the task scope (as if by invoking the [shutdown](#) method). It then waits for the threads executing any unfinished tasks to finish. If interrupted then

this method will continue to wait for the threads to finish before completing with the interrupt status set.

This method may only be invoked by the task scope owner. If the task scope is already closed then the owner invoking this method has no effect.

A `StructuredTaskScope` is intended to be used in a *structured manner*. If this method is called to close a task scope before nested task scopes are closed then it closes the underlying construct of each nested task scope (in the reverse order that they were created in), closes this task scope, and then throws `StructureViolationException`. Similarly, if this method is called to close a task scope while executing with `scoped value` bindings, and the task scope was created before the scoped values were bound, then `StructureViolationException` is thrown after closing the task scope. If a thread terminates without first closing task scopes that it owns then termination will cause the underlying construct of each of its open tasks scopes to be closed. Closing is performed in the reverse order that the task scopes were created in. Thread termination may therefore be delayed when the owner has to wait for threads forked in these task scopes to finish.

Specified by:

`close` in interface `AutoCloseable`

Throws:

`IllegalStateException` - thrown after closing the task scope if the owner did not invoke `join` after forking

`WrongThreadException` - if the current thread is not the owner

`StructureViolationException` - if a structure violation was detected

[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](#).

Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.

Copyright © 1993, 2023, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.

All rights reserved. Use is subject to [license terms](#) and the [documentation redistribution policy](#). [Modify Cookie Preferences](#). [Modify Ad Choices](#).