Module java.base
Package java.util.concurrent

Interface ExecutorService

All Superinterfaces:

Executor

All Known Subinterfaces:

ScheduledExecutorService

All Known Implementing Classes:

AbstractExecutorService, ForkJoinPool, ScheduledThreadPoolExecutor, ThreadPoolExecutor

---

```
public interface ExecutorService
extends Executor
```

An `Executor` that provides methods to manage termination and methods that can produce a `Future` for tracking progress of one or more asynchronous tasks.

An `ExecutorService` can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an `ExecutorService`. The `shutdown()` method will allow previously submitted tasks to execute before terminating, while the `shutdownNow()` method prevents waiting tasks from starting and attempts to stop currently executing tasks. Upon termination, an executor has no tasks actively executing, no tasks awaiting execution, and no new tasks can be submitted. An unused `ExecutorService` should be shut down to allow reclamation of its resources.

Method `submit` extends base method `Executor.execute(Runnable)` by creating and returning a `Future` that can be used to cancel execution and/or wait for completion. Methods `invokeAny` and `invokeAll` perform the most commonly useful forms of bulk execution, executing a collection of tasks and then waiting for at least one, or all, to complete. (Class `ExecutorCompletionService` can be used to write customized variants of these methods.)

The `Executors` class provides factory methods for the executor services provided in this package.

Usage Examples

Here is a sketch of a network service in which threads in a thread pool service incoming requests. It uses the preconfigured `Executors.newFixedThreadPool(int)` factory method:

```
class NetworkService implements Runnable {
  private final ServerSocket serverSocket;
  private final ExecutorService pool;

  public NetworkService(int port, int poolSize)
      throws IOException {
    serverSocket = new ServerSocket(port);
    pool = Executors.newFixedThreadPool(poolSize);
```

```
    }

    public void run() { // run the service
      try {
        for (;;) {
          pool.execute(new Handler(serverSocket.accept()));
        }
      } catch (IOException ex) {
        pool.shutdown();
      }
    }
  }

  class Handler implements Runnable {
    private final Socket socket;
    Handler(Socket socket) { this.socket = socket; }
    public void run() {
      // read and service request on socket
    }
  }
```

The following method shuts down an `ExecutorService` in two phases, first by calling `shutdown` to reject incoming tasks, and then calling `shutdownNow`, if necessary, to cancel any lingering tasks:

```
  void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // Disable new tasks from being submitted
    try {
      // Wait a while for existing tasks to terminate
      if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
        pool.shutdownNow(); // Cancel currently executing tasks
        // Wait a while for tasks to respond to being cancelled
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Pool did not terminate");
      }
    } catch (InterruptedException ex) {
      // (Re-)Cancel if current thread also interrupted
      pool.shutdownNow();
      // Preserve interrupt status
      Thread.currentThread().interrupt();
    }
  }
```

Memory consistency effects: Actions in a thread prior to the submission of a `Runnable` or `Callable` task to an `ExecutorService` *happen-before* any actions taken by that task, which in turn *happen-before* the result is retrieved via `Future.get()`.

Since:

1.5

Method Summary

All Methods Instance Methods Abstract Methods

Modifier and Type	Method	Description
boolean	**awaitTermination**(long timeout, **TimeUnit** unit)	Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
<T> **List**<**Future**<T>>	**invokeAll**(**Collection**<? extends **Callable**<T>> tasks)	Executes the given tasks, returning a list of Futures holding their status and results when all complete.
<T> **List**<**Future**<T>>	**invokeAll**(**Collection**<? extends **Callable**<T>> tasks, long timeout, **TimeUnit** unit)	Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.
<T> T	**invokeAny**(**Collection**<? extends **Callable**<T>> tasks)	Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.
<T> T	**invokeAny**(**Collection**<? extends **Callable**<T>> tasks, long timeout, **TimeUnit** unit)	Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.
boolean	**isShutdown**()	Returns `true` if this executor has been shut down.
boolean	**isTerminated**()	Returns `true` if all tasks have completed following shut down.
void	**shutdown**()	Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
List<**Runnable**>	**shutdownNow**()	Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

`Future<?>`	`submit(Runnable task)`	Submits a Runnable task for execution and returns a Future representing that task.
`<T> Future<T>`	`submit(Runnable task, T result)`	Submits a Runnable task for execution and returns a Future representing that task.
`<T> Future<T>`	`submit(Callable<T> task)`	Submits a value-returning task for execution and returns a Future representing the pending results of the task.

Methods declared in interface java.util.concurrent.Executor

`execute`

Method Details

shutdown

```
void shutdown()
```

Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Invocation has no additional effect if already shut down.

This method does not wait for previously submitted tasks to complete execution. Use `awaitTermination` to do that.

Throws:

`SecurityException` - if a security manager exists and shutting down this ExecutorService may manipulate threads that the caller is not permitted to modify because it does not hold `RuntimePermission("modifyThread")`, or the security manager's `checkAccess` method denies access.

shutdownNow

```
List<Runnable> shutdownNow()
```

Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

This method does not wait for actively executing tasks to terminate. Use `awaitTermination` to do that.

There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via `Thread.interrupt()`, so any task that fails to respond to interrupts may never terminate.

Returns:

list of tasks that never commenced execution

Throws:

`SecurityException` - if a security manager exists and shutting down this ExecutorService may manipulate threads that the caller is not permitted to modify because it does not hold `RuntimePermission("modifyThread")`, or the security manager's `checkAccess` method denies access.

isShutdown

```
boolean isShutdown()
```

Returns `true` if this executor has been shut down.

Returns:

`true` if this executor has been shut down

isTerminated

```
boolean isTerminated()
```

Returns `true` if all tasks have completed following shut down. Note that `isTerminated` is never `true` unless either `shutdown` or `shutdownNow` was called first.

Returns:

`true` if all tasks have completed following shut down

awaitTermination

```
boolean awaitTermination(long timeout,
                         TimeUnit unit)
                  throws InterruptedException
```

Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

Parameters:

`timeout` - the maximum time to wait

`unit` - the time unit of the timeout argument

Returns:

`true` if this executor terminated and `false` if the timeout elapsed before termination

Throws:

`InterruptedException` - if interrupted while waiting

submit

```
<T> Future<T> submit(Callable<T> task)
```

Submits a value-returning task for execution and returns a Future representing the pending results of the task. The Future's `get` method will return the task's result upon successful completion.

If you would like to immediately block waiting for a task, you can use constructions of the form `result = exec.submit(aCallable).get();`

Note: The `Executors` class includes a set of methods that can convert some other common closure-like objects, for example, `PrivilegedAction` to `Callable` form so they can be submitted.

Type Parameters:

`T` - the type of the task's result

Parameters:

`task` - the task to submit

Returns:

a Future representing pending completion of the task

Throws:

`RejectedExecutionException` - if the task cannot be scheduled for execution

`NullPointerException` - if the task is null

submit

```
<T> Future<T> submit(Runnable task,
                     T result)
```

Submits a Runnable task for execution and returns a Future representing that task. The Future's `get` method will return the given result upon successful completion.

Type Parameters:

`T` - the type of the result

Parameters:

`task` - the task to submit

`result` - the result to return

Returns:

a Future representing pending completion of the task

Throws:

`RejectedExecutionException` - if the task cannot be scheduled for execution

`NullPointerException` - if the task is null

submit

```
Future<?> submit(Runnable task)
```

Submits a Runnable task for execution and returns a Future representing that task. The Future's `get` method will return `null` upon *successful* completion.

Parameters:

`task` - the task to submit

Returns:

a Future representing pending completion of the task

Throws:

`RejectedExecutionException` - if the task cannot be scheduled for execution

`NullPointerException` - if the task is null

invokeAll

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
                       throws InterruptedException
```

Executes the given tasks, returning a list of Futures holding their status and results when all complete. `Future.isDone()` is `true` for each element of the returned list. Note that a *completed* task could have terminated either normally or by throwing an exception. The results of this method are undefined if the given collection is modified while this operation is in progress.

Type Parameters:

T - the type of the values returned from the tasks

Parameters:

`tasks` - the collection of tasks

Returns:

a list of Futures representing the tasks, in the same sequential order as produced by the iterator for the given task list, each of which has completed

Throws:

`InterruptedException` - if interrupted while waiting, in which case unfinished tasks are cancelled

`NullPointerException` - if tasks or any of its elements are `null`

`RejectedExecutionException` - if any task cannot be scheduled for execution

invokeAll

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                              long timeout,
                              TimeUnit unit)
                       throws InterruptedException
```

Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first. `Future.isDone()` is `true` for each element of the returned list. Upon return, tasks that have not completed are cancelled. Note that a *completed* task could have terminated either normally or by throwing an exception. The results of this method are undefined if the given collection is modified while this operation is in progress.

Type Parameters:

T - the type of the values returned from the tasks

Parameters:

`tasks` - the collection of tasks

`timeout` - the maximum time to wait

`unit` - the time unit of the timeout argument

Returns:

a list of Futures representing the tasks, in the same sequential order as produced by the iterator for the given task list. If the operation did not time out, each task will have completed. If it did time out, some of these tasks will not have completed.

Throws:

`InterruptedException` - if interrupted while waiting, in which case unfinished tasks are cancelled

`NullPointerException` - if tasks, any of its elements, or unit are `null`

`RejectedExecutionException` - if any task cannot be scheduled for execution

invokeAny

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
         throws InterruptedException,
                ExecutionException
```

Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do. Upon normal or exceptional return, tasks that have not completed are cancelled. The results of this method are undefined if the given collection is modified while this operation is in progress.

Type Parameters:

T - the type of the values returned from the tasks

Parameters:

`tasks` - the collection of tasks

Returns:

the result returned by one of the tasks

Throws:

`InterruptedException` - if interrupted while waiting

`NullPointerException` - if tasks or any element task subject to execution is `null`

`IllegalArgumentException` - if tasks is empty

`ExecutionException` - if no task successfully completes

`RejectedExecutionException` - if tasks cannot be scheduled for execution

invokeAny

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks,
                long timeout,
                TimeUnit unit)
         throws InterruptedException,
                ExecutionException,
                TimeoutException
```

Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses. Upon normal or exceptional return, tasks that have not completed are cancelled. The results of this method are undefined if the given collection is modified while this operation is in progress.

Type Parameters:

T - the type of the values returned from the tasks

Parameters:

`tasks` - the collection of tasks

`timeout` - the maximum time to wait

`unit` - the time unit of the timeout argument

Returns:

the result returned by one of the tasks

Throws:

`InterruptedException` - if interrupted while waiting

`NullPointerException` - if tasks, or unit, or any element task subject to execution is `null`

`TimeoutException` - if the given timeout elapses before any task successfully completes

`ExecutionException` - if no task successfully completes

`RejectedExecutionException` - if tasks cannot be scheduled for execution
