Java ExecutorService and Thread Pools Tutorial

Executors Framework

While it is easy to create one or two threads and run them, it becomes a problem when your application requires creating 20 or 30 threads for running tasks concurrently.

Also, it won't be exaggerating to say that large multi-threaded applications will have hundreds, if not thousands of threads running simultaneously. So, it makes sense to separate thread creation and management from the rest of the application.

Enter Executors, A framework for creating and managing threads. Executors framework helps you with -

1. **Thread Creation**: It provides various methods for creating threads, more specifically a pool of threads, that your application can use to run tasks concurrently.
2. **Thread Management**: It manages the life cycle of the threads in the thread pool. You don't need to worry about whether the threads in the thread pool are active or busy or dead before submitting a task for execution.
3. **Task submission and execution**: Executors framework provides methods for submitting tasks for execution in the thread pool, and also gives you the power to decide when the tasks will be executed. For example, You can submit a task to be executed now or schedule them to be executed later or make them execute periodically.

Java Concurrency API defines the following three executor interfaces that covers everything that is needed for creating and managing threads -

- **Executor** - A simple interface that contains a method called `execute()` to launch a task specified by a `Runnable` object.
- **ExecutorService** - A sub-interface of `Executor` that adds functionality to manage the lifecycle of the tasks. It also provides a `submit()` method whose overloaded versions can accept a `Runnable` as well as a `Callable` object. Callable objects are similar to Runnable except that the task specified by a Callable object can also return a value.

- **ScheduledExecutorService** - A sub-interface of `ExecutorService`. It adds functionality to schedule the execution of the tasks.

Apart from the above three interfaces, The API also provides an Executors class that contains factory methods for creating different kinds of executor services.

ExecutorService example

All right! let's dive into an example now to understand things better. In the following example, we first create an ExecutorService with a single worker thread, and then submit a task to be executed inside the worker thread.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorsExample {
    public static void main(String[] args) {
        System.out.println("Inside : " + Thread.currentThread().getName());

        System.out.println("Creating Executor Service...");
        ExecutorService executorService =
Executors.newSingleThreadExecutor();

        System.out.println("Creating a Runnable...");
        Runnable runnable = () -> {
            System.out.println("Inside : " +
Thread.currentThread().getName());
        };

        System.out.println("Submit the task specified by the runnable to the
executor service.");
        executorService.submit(runnable);
    }
}
# Output
Inside : main
Creating Executor Service...
Creating a Runnable...
Submit the task specified by the runnable to the executor service.
Inside : pool-1-thread-1
```

The above example shows how to create an executor service and execute a task inside the executor. We use the `Executors.newSingleThreadExecutor()` method to create an `ExecutorService` that uses a single worker thread for executing tasks. If a task is submitted for execution and the thread is currently busy executing another task, then the new task will wait in a queue until the thread is free to execute it.

If you run the above program, you will notice that the program never exits, because, the executor service keeps listening for new tasks until we shut it down explicitly.

Shutting down the ExecutorService

ExecutorService provides two methods for shutting down an executor -

- **shutdown()** - when `shutdown()` method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.

- **shutdownNow()** - this method interrupts the running task and shuts down the executor immediately.

Let's add shutdown code at the end of our program so that it exits gracefully -

```
System.out.println("Shutting down the executor");
executorService.shutdown();
```

ExecutorService example with multiple threads and tasks

In the earlier example, we created an ExecutorService that uses a single worker thread. But the real power of ExecutorService comes when we create a pool of threads and execute multiple tasks concurrently in the thread pool.

Following example shows how you can create an executor service that uses a thread pool and execute multiple tasks concurrently -

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ExecutorsExample {
    public static void main(String[] args) {
        System.out.println("Inside : " + Thread.currentThread().getName());

        System.out.println("Creating Executor Service with a thread pool of
Size 2");
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        Runnable task1 = () -> {
            System.out.println("Executing Task1 inside : " +
Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException ex) {
                throw new IllegalStateException(ex);
            }
        };

        Runnable task2 = () -> {
            System.out.println("Executing Task2 inside : " +
Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(4);
            } catch (InterruptedException ex) {
                throw new IllegalStateException(ex);
            }
        };

        Runnable task3 = () -> {
            System.out.println("Executing Task3 inside : " +
Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(3);
```

```
            } catch (InterruptedException ex) {
                throw new IllegalStateException(ex);
            }
        };


        System.out.println("Submitting the tasks for execution...");
        executorService.submit(task1);
        executorService.submit(task2);
        executorService.submit(task3);

        executorService.shutdown();
    }
}
# Output
Inside : main
Creating Executor Service with a thread pool of Size 2
Submitting the tasks for execution...
Executing Task2 inside : pool-1-thread-2
Executing Task1 inside : pool-1-thread-1
Executing Task3 inside : pool-1-thread-1
```

In the example above, we created an executor service with a fixed thread pool of size 2. A fixed thread pool is a very common type of thread pool that is frequently used in multi-threaded applications.

In a fixed thread-pool, the executor service makes sure that the pool always has the specified number of threads running. If any thread dies due to some reason, it is replaced by a new thread immediately.

When a new task is submitted, the executor service picks one of the available threads from the pool and executes the task on that thread. If we submit more tasks than the available number of threads and all the threads are currently busy executing the existing tasks, then the new tasks will wait for their turn in a queue.

Thread Pool

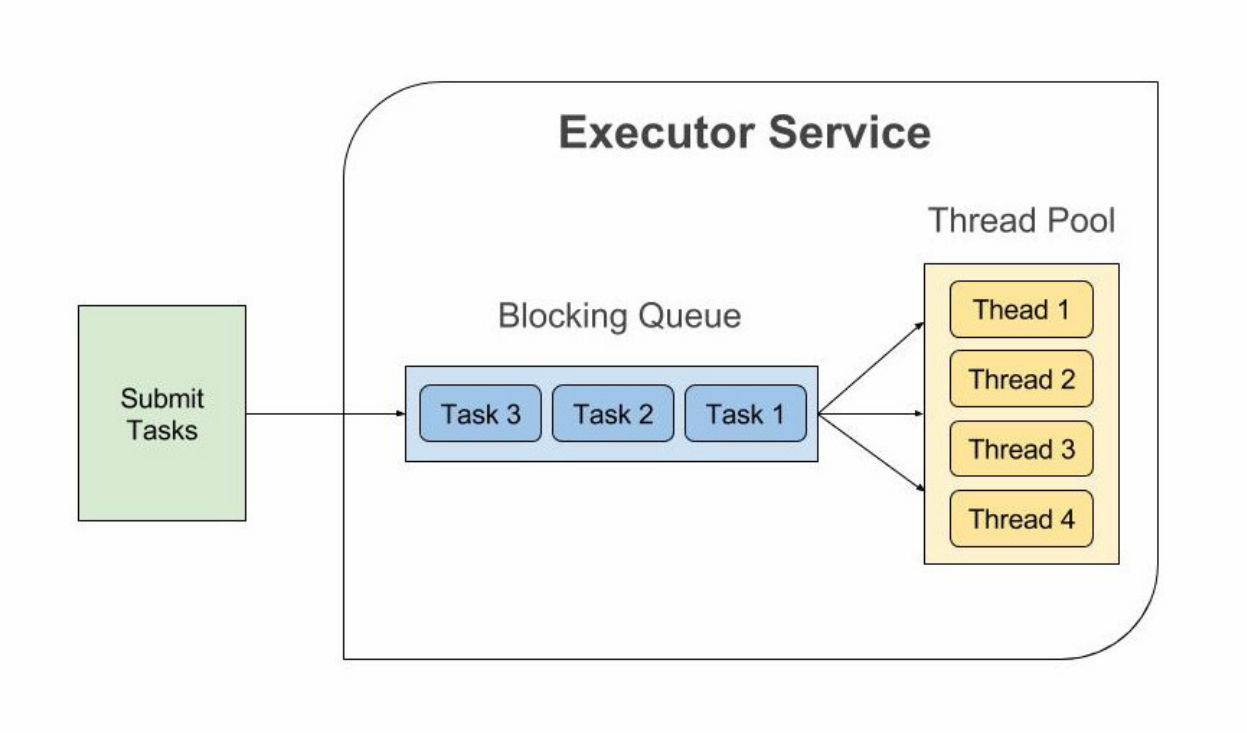
Most of the executor implementations use *thread pools* to execute tasks. A thread pool is nothing but a bunch of worker threads that exist separately from the `Runnable` or `Callable` tasks and is managed by the executor.

Creating a thread is an expensive operation and it should be minimized. Having worker threads minimizes the overhead due to thread creation because executor service has to create the thread pool only once and then it can reuse the threads for executing any task.

We already saw an example of a thread pool in the previous section called a fixed thread-pool.

Tasks are submitted to a thread pool via an internal queue called the ***Blocking Queue***. If there are more tasks than the number of active threads, they are inserted into the blocking queue for

waiting until any thread becomes available. If the blocking queue is full than new tasks are rejected.



ScheduledExecutorService example

ScheduledExecutorService is used to execute a task either periodically or after a specified delay.

In the following example, We schedule a task to be executed after a delay of 5 seconds -

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
        Runnable task = () -> {
          System.out.println("Executing Task At " + System.nanoTime());
        };

        System.out.println("Submitting task at " + System.nanoTime() + " to
be executed after 5 seconds.");
        scheduledExecutorService.schedule(task, 5, TimeUnit.SECONDS);

        scheduledExecutorService.shutdown();
    }
}
# Output
```

```
Submitting task at 2909896838099 to be executed after 5 seconds.
Executing Task At 2914898174612
```

`scheduledExecutorService.schedule()` function takes a `Runnable`, a delay value, and the unit of the delay. The above program executes the task after 5 seconds from the time of submission.

Now let's see an example where we execute the task periodically -

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsPeriodicExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);

        Runnable task = () -> {
          System.out.println("Executing Task At " + System.nanoTime());
        };

        System.out.println("scheduling task to be executed every 2 seconds
with an initial delay of 0 seconds");
        scheduledExecutorService.scheduleAtFixedRate(task, 0,2,
TimeUnit.SECONDS);
    }
}
# Output
scheduling task to be executed every 2 seconds with an initial delay of 0
seconds
Executing Task At 2996678636683
Executing Task At 2998680789041
Executing Task At 3000679706326
Executing Task At 3002679224212
.....
```

`scheduledExecutorService.scheduleAtFixedRate()` method takes a `Runnable`, an initial delay, the period of execution, and the time unit. It starts the execution of the given task after the specified delay and then executes it periodically on an interval specified by the period value.

Note that if the task encounters an exception, subsequent executions of the task are suppressed. Otherwise, the task will only terminate if you either shut down the executor or kill the program.