Java ExecutorService and Thread Pools Tutorial

Rajeev Kumar Singh • Java • Jun 26, 2017 • 8 mins read

Welcome to the third part of my tutorial series on Java concurrency. In this tutorial, we will learn how to manage threads in our application using executors and thread pools.

Executors Framework

In the previous tutorial, we learned how to create threads in Java by extending the `Thread` class or implementing the `Runnable` interface.

While it is easy to create one or two threads and run them, it becomes a problem when your

CALLICODER

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

CALLICODER

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. We'll learn about Callable in more detail, in the next blog post.
- **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.

CALLICODER

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

public class ExecutorsExample {
    public static void main(String[] ar
        System.out.println("Inside : "

        System.out.println("Creating Ex
        ExecutorService executorService

        System.out.println("Creating a
        Runnable runnable = () -> {
            System.out.println("Inside
        };

        System.out.println("Submit the
        executorService.submit(runnable
    }
}
```

```
# Output
Inside : main
Creating Executor Service...
Creating a Runnable...
Submit the task specified by the runnab
Shutting down the executor
Inside : pool-1-thread-1
```

CALLICODER

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.

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 e
```

CALLICODER

In the above 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.ExecutorSer
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ExecutorsExample {
    public static void main(String[] ar
        System.out.println("Inside : "

        System.out.println("Creating Ex
        ExecutorService executorService

        Runnable task1 = () -> {
            System.out.println("Executi
            try {
                TimeUnit.SECONDS.sleep(
            } catch (InterruptedExcepti
                throw new IllegalStateE
            }
```

CALLICODER

```
        Runnable task2 = () -> {
            System.out.println("Executi
            try {
                TimeUnit.SECONDS.sleep(
            } catch (InterruptedExcepti
                throw new IllegalStateE
            }
        };

        Runnable task3 = () -> {
            System.out.println("Executi
            try {
                TimeUnit.SECONDS.sleep(
            } catch (InterruptedExcepti
                throw new IllegalStateE
            }
        };


        System.out.println("Submitting
        executorService.submit(task1);
        executorService.submit(task2);
        executorService.submit(task3);

        executorService.shutdown();
    }
}
```

```
Creating Executor Service with a thread
Submitting the tasks for execution...
Executing Task1 inside : pool-1-thread-
Shutting down the ExecutorService...
Executing Task2 inside : pool-1-thread-
Executing Task3 inside : pool-1-thread-
Completed Task3 inside : pool-1-thread-
```

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

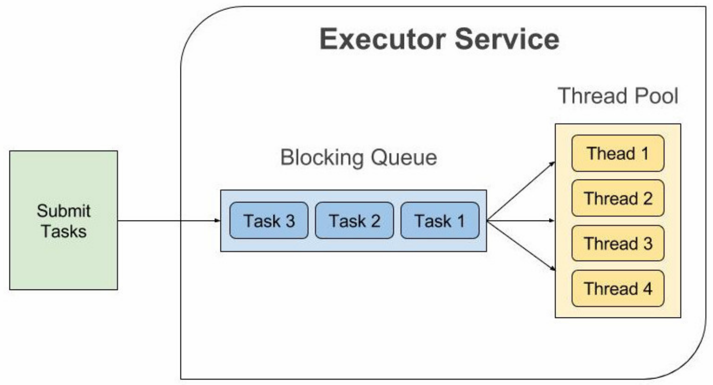
CALLICODER

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.



CALLICODER

ScheduledExecutorService is used to execute a task either p Search CalliCoder delay.

ng Spring Boot Node.js JavaFX

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.ScheduledEx
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsExample
    public static void main(String[] ar
        ScheduledExecutorService schedu
        Runnable task = () -> {
          System.out.println("Executing
        };

        System.out.println("Submitting
        scheduledExecutorService.schedu

        scheduledExecutorService.shutdo
    }
}
```

```
# Output
Submitting task at 2909896838099 to be
Executing Task At 2914898174612
```

Java Concurrency

`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.ScheduledEx
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsPeriodic
    public static void main(String[] ar
        ScheduledExecutorService schedu

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

        System.out.println("scheduling
        scheduledExecutorService.schedu
    }
}
```

```
# Output
scheduling task to be executed every 2
Executing Task At 2996678636683
```

CALLICODER

```
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.

Conclusion

In this blog post, we learned the basics of executors and thread pool. However, we have not yet covered all the features that executor service offers because for covering those features, we first need to understand two more topics - Callable and Future. We'll cover these topics in the next blog post.

All the code samples used in this tutorial can be found in my github repository. Please ask any doubts or clarifications in the comment section

CALLICODER