Java Callable and Future Tutorial

Rajeev Kumar Singh • Java • Jun 28, 2017 • 7 mins read

Welcome to the fourth part of my tutorial series on Java Concurrency. In earlier tutorials, we learned the basics of concurrency, threads, runnables and executor services. In this tutorial, we'll learn about Callable and Future.

Callable

In the previous tutorials, we used a `Runnable` object to define the tasks that are executed inside a thread. While defining tasks using `Runnable` is very convenient, it is limited by the fact that the tasks can not return a result.

Well, Java provides a `Callable` interface to define tasks that return a result. A `Callable` is similar to `Runnable` except that it can return a result and throw a checked exception.

`Callable` interface has a single method `call()` which is meant to contain the code that is executed by a thread. Here is an example of a simple Callable -

```
Callable<String> callable = new Callabl
    @Override
    public String call() throws Excepti
        // Perform some computation
        Thread.sleep(2000);
        return "Return some result";
    }
};
```

Note that with `Callable`, you don't need to surround `Thread.sleep()` by a try/catch block, because unlike Runnable, a Callable can throw a checked exception.

You can also use a lambda expression with Callable like this -

```
Callable<String> callable = () -> {
    // Perform some computation
```

```
};
```

Executing Callable tasks using ExecutorService and obtaining the result using Future

Just like `Runnable`, you can submit a `Callable` to an executor service for execution. But what about the Callable's result? How do you access it?

The `submit()` method of executor service submits the task for execution by a thread. However, it doesn't know when the result of the submitted task will be available. Therefore, it returns a special type of value called a `Future` which can be used to fetch the result of the task when it is available.

The concept of `Future` is similar to Promise in other languages like Javascript. It represents the result of a computation that will be completed at a later point of time in future.

Following is a simple example of Future and Callable -

CALLICODER

```
public class FutureAndCallableExample {
    public static void main(String[] ar
        ExecutorService executorService

        Callable<String> callable = ()
            // Perform some computation
            System.out.println("Entered
            Thread.sleep(2000);
            return "Hello from Callable
        };

        System.out.println("Submitting
        Future<String> future = executo

        // This line executes immediate
        System.out.println("Do somethin

        System.out.println("Retrieve th
        // Future.get() blocks until th
        String result = future.get();
        System.out.println(result);

        executorService.shutdown();
    }

}
```

```
# Output
```

CALLICODER

```
Retrieve the result of the future
Entered Callable
Hello from Callable
```

`ExecutorService.submit()` method returns immediately and gives you a Future. Once you have obtained a future, you can execute other tasks in parallel while your submitted task is executing, and then use `future.get()` method to retrieve the result of the future.

Note that, the `get()` method blocks until the task is completed. The `Future` API also provides an `isDone()` method to check whether the task is completed or not -

```
import java.util.concurrent.*;

public class FutureIsDoneExample {
    public static void main(String[] ar
        ExecutorService executorService

        Future<String> future = executo
            Thread.sleep(2000);
            return "Hello from Callable
        });

        while(!future.isDone()) {
            System.out.println("Task is
```

CALLICODER

```
        System.out.println("Task comple
        String result = future.get();
        System.out.println(result);

        executorService.shutdown();
    }
}
```

```
# Output
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task completed! Retrieving the result
Hello from Callable
```

Cancelling a Future

You can cancel a future using `Future.cancel()` method. It attempts to

CALLICODER

returns false.

The `cancel()` method accepts a boolean argument - `mayInterruptIfRunning`. If you pass the value `true` for this argument, then the thread that is currently executing the task will be interrupted, otherwise in-progress tasks will be allowed to complete.

You can use `isCancelled()` method to check if a task is cancelled or not. Also, after the cancellation of the task, `isDone()` will always true.

```
import java.util.concurrent.*;

public class FutureCancelExample {
    public static void main(String[] ar
        ExecutorService executorService

        long startTime = System.nanoTim
        Future<String> future = executo
            Thread.sleep(2000);
            return "Hello from Callable
        });

        while(!future.isDone()) {
            System.out.println("Task is
            Thread.sleep(200);
            double elapsedTimeInSec = (
```

CALLICODER

```
                future.cancel(true);
            }
        }

        System.out.println("Task comple
        String result = future.get();
        System.out.println(result);

        executorService.shutdown();
    }
}
```

```
# Output
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task completed! Retrieving the result
Exception in thread "main" java.util.co
        at java.util.concurrent.FutureT
        at java.util.concurrent.FutureT
        at FutureCancelExample.main(Fut
```

If you run the above program, it will throw an exception, because `future.get()` method throws `CancellationException` if the task is cancelled. We can handle this fact by checking

```
if(!future.isCancelled()) {
    System.out.println("Task completed!
    String result = future.get();
    System.out.println(result);
} else {
    System.out.println("Task was cancel
}
```

Adding Timeouts

The `future.get()` method blocks and waits for the task to complete. If you call an API from a remote service in the callable task and the remote service is down, then `future.get()` will block forever, which will make the application unresponsive.

To guard against this fact, you can add a timeout in the `get()` method -

```
future.get(1, TimeUnit.SECONDS);
```

The `future.get()` method will throw a `TimeoutException` if the task is not completed within the specified time.

invokeAll

CALLICODER

You can execute multiple tasks by passing a collection of Callables to the `invokeAll()` method. The `invokeAll()` returns a list of Futures. Any call to `future.get()` will block until all the Futures are complete.

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.*;

public class InvokeAllExample {
    public static void main(String[] ar
        ExecutorService executorService

        Callable<String> task1 = () ->
            Thread.sleep(2000);
            return "Result of Task1";
        };

        Callable<String> task2 = () ->
            Thread.sleep(1000);
            return "Result of Task2";
        };

        Callable<String> task3 = () ->
            Thread.sleep(5000);
            return "Result of Task3";
        };
```

```
        List<Future<String>> futures =

        for(Future<String> future: futu
            // The result is printed on
            System.out.println(future.g
        }

        executorService.shutdown();
    }
}
```

```
# Output
Result of Task1
Result of Task2
Result of Task3
```

In the above program, the first call to `future.get()` statement blocks until all the futures are complete. i.e. the results will be printed after 5 seconds.

invokeAny

Submit multiple tasks and wait for any one of them to complete

The `invokeAny()` method accepts a collection of `Callables` and returns the result of the

CALLICODER

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.*;

public class InvokeAnyExample {
    public static void main(String[] ar
        ExecutorService executorService

        Callable<String> task1 = () ->
            Thread.sleep(2000);
            return "Result of Task1";
        };

        Callable<String> task2 = () ->
            Thread.sleep(1000);
            return "Result of Task2";
        };

        Callable<String> task3 = () ->
            Thread.sleep(5000);
            return "Result of Task3";
        };

        // Returns the result of the fa
        String result = executorService

        System.out.println(result);
```

```
}
```

```
# Output
Result of Task2
```

Conclusion

You can find the all the code snippets used in this tutorial in my github repository. I encourage you to fork the repo and practice the programs yourself.

Don't forget to check out the next post in this tutorial series for learning about various issues related to concurrent programs and how to avoid them.

Thank you for reading. Please ask any questions in the comment section below.
