(http://baeldung.com)

A Guide to the Java ExecutorService

Last modified: January 26, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

ExecutorService (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) is a framework provided by the JDK which simplifies the execution of tasks in asynchronous mode. Generally speaking, *ExecutorService* automatically provides a pool of threads and API for assigning tasks to it.

Further reading:

Guide to the

Overview of the

Guide to

Fork/Join Framework in Java (http://www.baeldung.com/java-fork-join)

An intro to the fork/join framework presented in Java 7 and the tools to help speed up parallel processing by attempting to use all available processor cores.

Read more (http://www.baeldung.com/java-fork-join) →

java.util.concurrent (http://www.baeldung.com/java-concurrent)

Discover the content of the java.util.concurrent package.

Read more (http://www.baeldung.com/java-util-concurrent) →

java.util.concurrent.Locks (http://www.baeldung.com/java-concurrent-locks)

In this article, we explore various implementations of the Lock interface and the newly introduced in Java 9 StampedLock class.

Read more (http://www.baeldung.com/java-concurrent-locks) →

2. Instantiating *ExecutorService*

2.1. Factory Methods of the *Executors* Class

The easiest way to create *ExecutorService* is to use one of the factory methods of the *Executors* class.

For example, the following line of code will create a thread-pool with 10 threads:

```
1 ExecutorService executor = Executors.newFixedThreadPool(10);
```

The are several other factory methods to create predefined *ExecutorService* that meet specific use cases. To find the best method for your needs, consult Oracle's official documentation (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html).

2.2. Directly Create an *ExecutorService*

Because *ExecutorService* is an interface, an instance of any its implementations can be used. There are several implementations to choose from in the *java.util.concurrent (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html)* package or you can create your own.

For example, the *ThreadPoolExecutor* class has a few constructors which can be used to configure an executor service and its internal pool.

```
ExecutorService executorService =
  new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
  new LinkedBlockingQueue<Runnable>());
```

You may notice that the code above is very similar to the source code (http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/Executors.java#Executors.newSingleThreadExecutor%28%29) of the factory method *newSingleThreadExecutor().* For most cases, a detailed manual configuration isn't necessary.

3. Assigning Tasks to the *ExecutorService*

ExecutorService can execute *Runnable* and *Callable* tasks. To keep things simple in this article, two primitive tasks will be used. Notice that lambda expressions are used here instead of anonymous inner classes:

```
Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Callable<String> callableTask = () -> {
    TimeUnit.MILLISECONDS.sleep(300);
    return "Task's execution";
};

List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(callableTask);
callableTasks.add(callableTask);
callableTasks.add(callableTask);
```

Tasks can be assigned to the *ExecutorService* using several methods, including *execute()*, which is inherited from the *Executor* interface, and also *submit(), invokeAny(), invokeAll().*

The ***execute()*** method is *void,* and it doesn't give any possibility to get the result of task's execution or to check the task's status (is it running or executed).

```
executorService.execute(runnableTask);
```

submit() submits a *Callable* or a *Runnable* task to an *ExecutorService* and returns a result of type *Future*.

```
Future<String> future =
  executorService.submit(callableTask);
```

invokeAny() assigns a collection of tasks to an *ExecutorService,* causing each to be executed, and returns the result of a successful execution of one task (if there was a successful execution).

```
1 String result = executorService.invokeAny(callableTasks);
```

invokeAll() assigns a collection of tasks to an *ExecutorService,* causing each to be executed, and returns the result of all task executions in the form of a list of objects of type *Future.*

```
1 List<Future<String>> futures = executorService.invokeAll(callableTasks);
```

Now, before going any further, two more things must be discussed: shutting down an *ExecutorService* and dealing with *Future* return types.

4. Shutting Down an *ExecutorService*

In general, the *ExecutorService* will not be automatically destroyed when there is not task to process. It will stay alive and wait for new work to do.

In some cases this is very helpful; for example, if an app needs to process tasks which appear on an irregular basis or the quantity of these tasks is not known at compile time.

On the other hand, an app could reach its end, but it will not be stopped because a waiting *ExecutorService* will cause the JVM to keep running.

To properly shut down an *ExecutorService*, we have the *shutdown()* and *shutdownNow()* APIs.

The ***shutdown()*** method doesn't cause an immediate destruction of the *ExecutorService.* It will make the *ExecutorService* stop accepting new tasks and shut down after all running threads finish their current work.

```
1 executorService.shutdown();
```

The ***shutdownNow()*** method tries to destroy the *ExecutorService* immediately, but it doesn't guarantee that all the running threads will be stopped at the same time. This method returns a list of tasks which are waiting to be processed. It is up to the developer to decide what to do with these tasks.

```
1 List<Runnable> notExecutedTasks = executorService.shutDownNow();
```

One good way to shut down the *ExecutorService* (which is also recommended by Oracle (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html)) is to use both of these methods combined with the ***awaitTermination()*** method. With this approach, the *ExecutorService* will first stop taking new tasks, the wait up to a specified period of time for all tasks to be completed. If that time expires, the execution is stopped immediately:

```
executorService.shutdown();
try {
    if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {
        executorService.shutdownNow();
    }
} catch (InterruptedException e) {
    executorService.shutdownNow();
}
```

5. The *Future* Interface

The *submit()* and *invokeAll()* methods return an object or a collection of objects of type *Future*, which allows us to get the result of a task's execution or to check the task's status (is it running or executed).

The *Future* interface provides a special blocking method *get()* which returns an actual result of the *Callable* task's execution or *null* in the case of *Runnable* task. Calling the *get()* method while the task is still running will cause execution to block until the task is properly executed and the result is available.

```
Future<String> future = executorService.submit(callableTask);
String result = null;
try {
    result = future.get();
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

With very long blocking caused by the *get()* method, an application's performance can degrade. If the resulting data is not crucial, it is possible to avoid such a problem by using timeouts:

```
String result = future.get(200, TimeUnit.MILLISECONDS);
```

If the execution period is longer than specified (in this case 200 milliseconds), a *TimeoutException* will be thrown.

The *isDone()* method can be used to check if the assigned task is already processed or not.

The *Future* interface also provides for the cancellation of task execution with the *cancel()* method, and to check the cancellation with *isCancelled()* method:

```
boolean canceled = future.cancel(true);
boolean isCancelled = future.isCancelled();
```

6. The *ScheduledExecutorService* Interface

The *ScheduledExecutorService* runs tasks after some predefined delay and/or periodically. Once again, the best way to instantiate a *ScheduledExecutorService* is to use the factory methods of the *Executors* class.

For this section, a *ScheduledExecutorService* with one thread will be used:

```
ScheduledExecutorService executorService = Executors
  .newSingleThreadScheduledExecutor();
```

To schedule a single task's execution after a fixed delay, us the *scheduled()* method of the *ScheduledExecutorService*. There are two *scheduled()* methods that allow you to execute *Runnable* or *Callable* tasks:

```
Future<String> resultFuture =
  executorService.schedule(callableTask, 1, TimeUnit.SECONDS);
```

The *scheduleAtFixedRate()* method lets execute a task periodically after a fixed delay. The code above delays for one second before executing *callableTask*.

The following block of code will execute a task after an initial delay of 100 milliseconds, and after that, it will execute the same task every 450 milliseconds. If the processor needs more time to execute an assigned task than the *period* parameter of the *scheduleAtFixedRate()* method, the *ScheduledExecutorService* will wait until the current task is completed before starting the next:

```
Future<String> resultFuture = service
  .scheduleAtFixedRate(runnableTask, 100, 450, TimeUnit.MILLISECONDS);
```

If it is necessary to have a fixed length delay between iterations of the task, *scheduleWithFixedDelay()* should be used. For example, the following code will guarantee a 150-millisecond pause between the end of the current execution and the start of another one.

```
service.scheduleWithFixedDelay(task, 100, 150, TimeUnit.MILLISECONDS);
```

Accord[illegible] and *scheduleWithFixedDelay()* method contracts, period execut[illegible]ation of the *ExecutorService* or if an exception is thrown[illegible]

7. *Ex[illegible]k/Join*

After th[illegible]s decided that the *ExecutorService* framework should be repl[illegible]is is not always the right decision, however. Despite the sim[illegible]rformance gains associated with fork/join, there is also a reduc[illegible]ntrol over concurrent execution.

Execu 4? ɔility to control the number of generated threads and the gra ecuted by separate threads. The best use case for *ExecutorService* is the processing of independent tasks, such as transactions or requests according to the scheme "one thread for one task."

In contrast, according to Oracle's documentation (https:// /essential/concurrency/forkjoin.html), fork/join was designe broken into smaller pieces recursively.

8. Conclusion

Even despite the relative simplicity of *ExecutorService*, there are a few common pitfalls. Let's summarize them:

Keeping an unused *ExecutorService* alive: There is a detailed explanation in section 4 of this article about how to shut down an *ExecutorService*;

Wrong thread-pool capacity while using fixed length thread-pool: It is very important to determine how many threads the application will need to execute tasks efficiently. A thread-pool that is too large will cause unnecessary overhead just to create threads which mostly will be in the waiting mode. Too few can make an application seem unresponsive because of long waiting periods for tasks in the queue;

Calling a *Future*'s *get()* method after task cancellation: An attempt to get the result of an already canceled task will trigger a *CancellationException.*

Unexpectedly-long blocking with *Future*'s *get()* method: Timeouts should be used to avoid unexpected waits.

The code for this article is available in a GitHub repository (https://github.com/eugenp/tutorials/tree/master/core-java-concurrency).