Theory: Callable and Future

35 minutes 0 / 5 problems solved

§1. The Callable interface

Sometimes you need not only to execute a task in an executor but also to return a result of this task to the calling code. It is possible but an inconvenient thing to do with `Runnable`'s.

In order to simplify it, an executor supports another class of tasks named `Callable` that returns the result and may throw an exception. This interface belongs to the `java.util.concurrent` package. Let's take a look at this.

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

As you can see, it is a generic interface where the type parameter `V` determines the type of a result. Since it is a **functional interface**, we can use it together with lambda expressions and method references as well as implementing classes.

Here is a `Callable` that emulates a long-running task and returns a number that was "calculated".

```
Callable<Integer> generator = () -> {
    TimeUnit.SECONDS.sleep(5);
    return 700000;
};
```

The same code can be rewritten using inheritance that is more boilerplate than the lambda.

§2. Submitting a Callable and obtaining a Future

When we submit a `Callable` to executor service, it cannot return a result directly since the `submit` method does not wait until the task completes. Instead, an executor returns a special object called `Future` that wraps the actual result that may not even exist yet. This object represents the result of an asynchronous computation (task).

```
ExecutorService executor = Executors.newSingleThreadExecutor();

Future<Integer> future = executor.submit(() -> {
    TimeUnit.SECONDS.sleep(5);
    return 700000;
};
```

Until the task completes, the actual result is not present in the `future`. To check it, there is a method `isDone()`. Most likely, it will return `false` if you will call it immediately after obtaining a new `future`.

```
System.out.println(future.isDone()); // most likely it is false
```

§3. Getting the actual result of a task

The result can only be retrieved from a *future* by using the `get` method.

```
int result = future.get();
```

It returns the result when the computation has completed, or block the current thread and waits for the result. This method may throw two checked exceptions: `ExecutionException` and `InterruptedException` which we omit here

Current topic:

Callable and Future

Topic depends on:

- ✕ Functional interfaces
- ✕ List
- ✕ Interruptions
- ✕ Executors

Table of contents:

for brevity.

If a submitted task executes an infinite loop or waits for an external resource for too long, a thread that invokes `get` will be blocked all this time. To prevent this, there is also an overloaded version of `get` with a waiting timeout.

```
int result = future.get(10, TimeUnit.SECONDS); // it blocks the current thread
```

In this case, the calling thread waits for 10 seconds most for the computation to complete. If the timeout ends, the method throws `TimeoutException`.

§4. Cancelling a task

The `Future` class provides an instance method named `cancel` that attempts to cancel the execution of a task. This method is more complicated than it might seem at the first look.

An attempt will fail if the task has already completed, has already been canceled or could not be canceled for some other reason. If successful, and if this task has not already started when the method is invoked, it will never run.

The method takes a `boolean` parameter that determines whether the thread executing this task should be interrupted in an attempt to stop the task (in other words, whether to stop already running task or not).

```
future1.cancel(true);  // try to cancel even if the task is executing now
future2.cancel(false); // try to cancel only if the task is not executing
```

Since passing `true` involves interruptions, the cancelation of an executing task is guaranteed only if it handles `InterruptedException` correctly and checks the flag `Thread.currentThread().isInterrupted()`.

If someone invokes `future.get()` at a successfully canceled task, the method throws an unchecked `CancellationException`. If you do not want to deal with it, you may check whether a task was canceled by invoking `isCancelled()`.

§5. The advantage of using Callable and Future

The approach we are learning here allows us to do something useful between obtaining a `Future` and getting the actual result. In this time interval, we can submit several tasks to an executor, and only after that wait for all results to aggregate.

```
ExecutorService executor = Executors.newFixedThreadPool(4);

Future<Integer> future1 = executor.submit(() -> {
    TimeUnit.SECONDS.sleep(5);
    return 700000;
});

Future<Integer> future2 = executor.submit(() -> {
    TimeUnit.SECONDS.sleep(5);

    return 900000;

});



int result = future1.get() + future2.get(); // waiting for both results



System.out.println(result); // 1600000
```

If you have a modern computer, these tasks may be executed in parallel.

§6. Methods invokeAll and invokeAny

In addition to all features described above, there are two useful methods for submitting batches of `Callable` to an executor.

- `invokeAll` accepts a prepared collection of callables and returns a collection of *futures*;
- `invokeAny` also accepts a collection of callables and returns the result (not a *future*!) of one that has completed successfully.

Both methods also have overloaded versions that accept a timeout of execution that is often needed in real life.

Suppose that we need to calculate several numbers in separated tasks and then sum up the numbers in the `main` thread. It is easy to do by using `invokeAll` method.

```
ExecutorService executor = Executors.newFixedThreadPool(4);
List<Callable<Integer>> callables =
        List.of(() -> 1000, () -> 2000, () -
> 1500); // three "difficult" tasks

List<Future<Integer>> futures = executor.invokeAll(callables);
int sum = 0;
for (Future<Integer> future : futures) {
   sum += future.get(); // blocks on each future to get a result
}

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

If your version of Java is 8 rather than 9+ replace `List.of(...)` with `Arrays.asList(...)`. If you know Stream API (Java 8) and would like to practice it, you may rewrite this code using it.

§7. Summary

Let's summarize the information about `Callable` and `Future`.

To get a result of an asynchronous task executed in `ExecutorService` you have to execute three steps:

1. create an object representing a `Callable` task;
2. submit the task in `ExecutorService` and obtain a `Future`;
3. invoke `get` to receive the actual result when you need it.

Using `Future` allows us not to block the current thread until we do want to receive a result of a task. It is also possible to start multiple tasks and then get all results to aggregate them in the current thread. In addition to making your program more responsive, it will speed up your calculations if your computer supports parallel execution of threads.

You may also use methods `isDone`, `cancel` and `isCancelled` of a `future`. But be careful with exception handling when using them. Unfortunately, we cannot give all possible recipes and best practices within the lesson, but they will come with more experience. The main thing, especially in multi-threaded programming, is to read the documentation.

Report a typo

56 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (4) Hints (0) Useful links (0) Show discussion