Home » Core Java » util » concurrent » CompletableFuture » Java CompletionStage and CompletableFuture Example

ABOUT NAWAZISH KHAN

I am Nawazish, graduated in Electrical Engineering, in 2007. I work as a Senior Software Engineer with GlobalLogic India Ltd (Banglore) in the Telecom/Surveillance domain. A Java Community Process (JCP) member with an unconditional love (platform, technology, language, environment etc does not matter) for computer programming. Extremely interested programming multi-threaded applications, IoT devices (on top of JavaME) and application containers. The latest language of interest being Google Go; and Neo4j as the NoSQL Graph Database solution.

Java CompletionStage and CompletableFuture Example

Posted by: Nawazish Khan in CompletableFuture February 5th, 2016 0 1582 Views

1. Introduction

Java JDK8 included the big fat interface called

```
CompletionStage
```

in the

```
java.util.concurrent
```

package. The same package also contains

```
CompletableFuture
```

which is a library implementation of

```
CompletionStage
```

. In this post we would see how

```
CompletionStage
```

and

```
CompletableFuture
```

provide piped asynchronous API thus enhancing reactive programming support in Java at the platform level.

Since we are talking about JDK8 APIs, this post assumes familiarity with Lambda Expressions, Default Methods and Functional Interfaces.

2. CompletionStage And CompletableFuture Primer

2.1 CompletionStage

CompletionStage is an interface which abstracts units or blocks of computation which may or may not be asynchronous. It is important to realize that multiple CompletionStages, or in other words, units of works, can be piped together so that:

- The "completion" of the task on one "stage" could trigger computation of some other CompletionStage.
- The exceptional completion of one CompletionStage trigger computation of some other CompletionStage.
- The completion of "any" CompletionStage may trigger the computation of some other CompletionStage.
- The completion of "both" CompletionStage may trigger the computation of some other CompletionStage.
- The completion of "all" CompletionStage may trigger the computation of some other CompletionStage.

So on and so forth. Two things are important to notice: firstly, CompletionStage can abstract an asynchronous task and secondly a CompletionStage's outcome – asynchronous outcome – can be piped to trigger computation of some other dependent CompletionStage which could further trigger some other dependent CompletionStage, so on and so forth; and this lays the foundation of a reactive result processing which can have a valid use-case in virtually any area, from Edge Nodes to Gateways to Clients to Enterprise Apps to Cloud Solutions! Furthermore, potentially, this reduces superfluous polling checks for the availability of result and/or blocking calls on futuristic results. We will explain most of these propositions shortly through examples.

2.2 CompletableFuture

CompletableFuture is a concrete implementation of a CompletionStage and it also implements the java.util.concurrent.Future interface as well. This is the class which models a task (which may or may not be asynchronous) and exposes various methods to interact with the task; for instance, we have methods to check if the task has completed; whether it has completed exceptionally; we even have APIs to chain dependencies between multiple tasks; cancelling uncompleted tasks, so on and so forth. We would be looking into some of these APIs soon.

3. CompletableFuture programming model

A

```
CompletableFuture
```

can be instantiated and related methods can be called upon it, and we will see this in action in the subsequent section. However, there are convenient, static overloaded factory methods which provides further flexibility so that rather than worrying about harnessing

```
CompletableFuture
```

for a task, we can just concentrate on the task itself. I will explain this in a bit, but lets quickly have a look at the overloaded factory methods that I am talking about:

CompletableFuture supplyAsync() API

```
public static CompletableFuture supplyAsync(Supplier supplier)
public static CompletableFuture supplyAsync(Supplier supplier, Executor executor)
```

```
java.util.function.Supplier
```

is a functional interface which accepts nothing and "supplies" an output. The

```
supplyAsync()
```

API expects that a result-producing task be wrapped in a

```
Supplier
```

instance and handed over to the

```
supplyAsync()
```

method, which would then return a

```
CompletableFuture
```

representing this task. This task would, by default, be executed with one of the threads from the standard

```
java.util.concurrent.ForkJoinPool (public static ForkJoinPool commonPool())
```

.

However, we can also provide custom thread pool by passing a

```
java.util.concurrent.Executor
```

instance and as such the

```
Supplier
```

tasks would be scheduled on threads from this

```
Executor
```

instance.

So to sum up the, the easiest way of using

```
CompletableFuture
```

API is to wrap the task you want to execute in a

```
Supplier
```

– you may additionally supply an

```
Executor
```

service as needed – and hand it over to the

`supplyAsync()`

method which would return you the

`CompletableFuture`

!

There is yet another variant API available for retrieving a

`CompletableFuture`

. Notice that while discussing

`supplyAsync()`

I wrote that this is to be used when task would be result-bearing, in other words, when we expect the task to return back some output. However, in all cases where the task might not return any output, we may use the

`runAsyn()`

API, instead:

CompletableFuture runAsync() API

```java
public static CompletableFuture runAsync(Runnable runnable)
public static CompletableFuture runAsync(Runnable runnable, Executor executor)
```

Notice that

`runAsync()`

expects a java.lang.Runnable instance, and we know that

`Runnable.run()`

does not return any result! This is the reason that the returned

`CompletableFuture`

type erases itself to Void type.

4. Why the Name

A

`CompletableFuture`

can be instantiated through its no-arg constructor. And then we can manually provide a Runnable instance to a custom thread; and then

`CompletableFuture`

API provides

`complete`

method using which the

`CompletableFuture`

can be manually completed:

How to manually complete a CompletableFuture

```java
//1. Why named CompletableFuture?
        CompletableFuture completableFuture1 = new CompletableFuture();
        new Thread (()-> {
            try {
                Thread.sleep(4000L);
            } catch (Exception e) {
                completableFuture1.complete(-100.0);
            }
            /*
             * we can manually "complete" a CompletableFuture!!
             * this feature is not found with the classical Future interface
             */
            completableFuture1.complete(100.0);
        },"CompFut1-Thread").start();

        System.out.println("ok...waiting at: "+new Date());
        System.out.format("compFut value and received at: %f, %s \n", completableFuture1.join(), new Date());
```

5. Chaining multiple CompletableFutures: The Either Construct

The flexibility of asynchronous task processing actually comes by the virtue of chaining multiple tasks in a particular order, such that (asynchronous) completion of one CompletableFuture Task might fire asynchronous execution of another separate task:

Creating Either-Or dependency between different CompletableFutures

```
//2. chaining multiple CompletionStages dependencies - the "either" construct
        /**
         * A CompletionStage may have either/or completion dependency with
         * other CompletionStages: In the following snippet, completableFutureForAcptEither
         * depends on the completion of either CompletableFuture2 or CompletableFuture3
         */

        //We will create an ExecutorService rather than depending on ForkJoinCommonPool
        ExecutorService exec = Executors.newCachedThreadPool();

        CompletableFuture completableFuture2 =
CompletableFuture.supplyAsync(TaskSupplier::getSomeArbitraryDouble,exec);
        /*
         * we made TaskSupplier.getSomeArbitraryDouble to delay for 5s to bring asynchrony
         * with task wrapped within CompletableFuture3 (which we would be delaying for 3s)
         * If Operating System does not do schedule these tasks contrary to our expectations,
         * then CompletableFuture3 would complete before completableFuture2.
         */

        CompletableFuture completableFuture3 =
CompletableFuture.supplyAsync(TaskSupplier::getAnotherArbitraryDouble, exec);

        CompletableFuturecompletableFutureForAcptEither =
completableFuture2.acceptEitherAsync(completableFuture3, (val)-> {
                                    System.out.println("val: "+val);
                                }, exec);
```

6. Chaining multiple CompletableFutures: The One-After-The-Other Construct

I believe that the real reactive programming paradigm is provided by

```
CompletableFuture
```

APIs like

```
public CompletableFuture thenCompose(Function fn)
```

. In spirit, we allow a task to execute asynchronously and when its result is ready, we use it or fire yet another asynchronous task separately. The

```
thenCompose()
```

method helps in doing all of this. This method takes a

```
java.util.function.Function
```

, which accepts the result of this CompletableFuture, which may be processed as required and then returns a new

```
CompletableFuture
```

. Similarly this returned

```
CompletableFuture
```

can again be chained for firing some other

```
CompletableFuture
```

. However, note that if any of the

```
CompletableFuture
```

completes exceptionally then all subsequent dependent

```
CompletableFuture
```

s would complete with

```
java.util.concurrent.CompletionException
```

.

Creating sequential dependencies between CompletableFutures

```
//3. Chaining multiple CompletableFutures - one-after-the-other construct
        /*
         * We can chain various CompletableFutures one after the other provided
         * that the depending CompletableFuture completes normally.
         * The following snippet would clarify the construct.
         * In this example,completableFuture5 waits for the completion of
         * completableFuture4, as completableFuture5 would execute accordingly
         * depending on the outcome of completableFuture4
         */

CompletableFuture completableFuture4 = CompletableFuture.supplyAsync(TaskSupplier::getValueForCompletableFuture4, exec);
CompletableFuture completableFuture5 = completableFuture4.thenComposeAsync((compFut4)->{
            if (compFut4 == 100){
            CompletableFuture  compFut = new CompletableFuture();
            compFut.complete(1D);
            return compFut;
            }
               else if(compFut4 == 50){
                CompletableFuture  compFutt = new CompletableFuture();
                            compFutt.complete(0D);
                            return compFutt;
                        }
                          return null;
                    },exec);

        System.out.println("completableFuture5: "+completableFuture5.join());
```

7. What happens if a CompletableFuture Task completes exceptionally

```
CompletableFuture
```

API provides the flexibility of handling situations when one asynchronous task completes exceptionally. The API

```
public CompletableFuture exceptionally(Function fn)
```

comes handy towards this purpose. Basically method

```
exceptionally()
```

returns another

```
CompletableFuture
```

; now if the current

```
CompletableFuture
```

has completed its execution normally then the returned

```
CompletableFuture
```

(from

```
exceptionally()
```

method) would also complete with the same value; however, if the current

```
CompletableFuture
```

completes exceptionally then the

```
java.lang.Throwable
```

exception (which triggered the exceptional completion of the current

```
CompletableFuture
```

) is passed as an argument to the

```
java.util.function.Function
```

which would be executed to complete the returned

```
CompletableFuture
```

. In the code snippet below, I am checking if the

```
Throwable
```

returned is not null, and in such a case, I am logging the exceptional message (obviously, based on the application requirements, a lot of other things could have been done).

The following code snippet explains it emphasizing on the after-effects on any dependent

```
CompletableFuture
```

:

How to handle exceptional completion of CompletableFutures

```
//4. CompletableFuture chaining when the depending CompletableFuture completes exceptionally.
        CompletableFuture completableFuture6
                                    =
CompletableFuture.supplyAsync(TaskSupplier::throwRuntimeException);

        completableFuture6.exceptionally((throwable)->{
            if (throwable!=null){
                System.out.println("Exception thrown with message: "+throwable.getMessage());
                return null;
            }
            else
                return completableFuture6.join();
        });
```

8. Cancelling CompletableFuture Tasks

`CompletableFuture`

derives its cancellation policy from the classic

`Future`

interface and as such the semantics of cancelling a

`CompletableFuture`

task does not change.

`CompletableFuture`

exposes convenience API for cancelling a not-yet-completed task; the API is

`public boolean cancel(boolean mayInterruptIfRunning)`

.

As mentioned earlier, a

`CompletableFuture`

task can be cancelled only when it has not yet completed, implying that either (i) it wasn't yet scheduled for execution or (ii) it is currently under execution (and hasn't yet completed its execution). In both these situations that task can be cancelled. Such a cancellation accompanies tasks with

`java.util.concurrent.CancellationException`

such that calling task state retrieval methods like

`join()`

and

`get()`

would throw

`CancellationException`

. And it does not ends there, any subsequent dependent

`CompleteableFutures`

(remember CompletableFutures chaining from section 4. and 5.) would also complete with

`CancellationException`

It is also noteworthy that if a

`CompletableFuture`

task has completed, either normally or exceptionally, then cancelling it would be no-ops and the

`cancel()`

method would return with a boolean

`false`

.

Cancelling CompletableFuture

```
//5. CompletableFuture, if not already complete, can be cancelled with a relevant Exception
        CompletableFuture completableFuture7
                                    =
CompletableFuture.supplyAsync(TaskSupplier::cancelThisTask);
        boolean isCancelled = completableFuture7.cancel(true);
        System.out.println("Is completableFuture7 cancelled: "+isCancelled);
        System.out.println("Is completableFuture7 completed with exception:
"+completableFuture7.isCompletedExceptionally());
        /*
```

```
         * we know that completableFuture7 was cancelled and thus retrieving its state would
         * result in throwing of java.util.concurrent.CancellationException
        */
        System.out.println("Whats the result of task completableFuture7:
"+completableFuture7.join());
```

9. Conclusion

The flexibility of chaining multiple

```
CompletableFutures
```

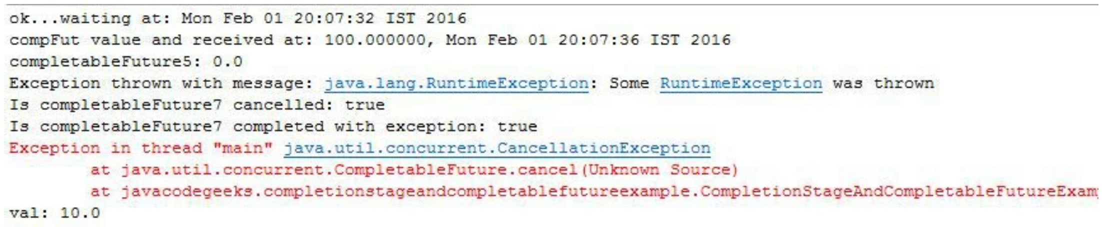
such that the completion of one triggers execution of another

```
CompletableFuture
```

; this opens up the paradigm of reactive programming in Java. Now there is no blocking call like

```
Future.get()
```

to retrieve the result of the future Task.



```
ok...waiting at: Mon Feb 01 20:07:32 IST 2016
compFut value and received at: 100.000000, Mon Feb 01 20:07:36 IST 2016
completableFuture5: 0.0
Exception thrown with message: java.lang.RuntimeException: Some RuntimeException was thrown
Is completableFuture7 cancelled: true
Is completableFuture7 completed with exception: true
Exception in thread "main" java.util.concurrent.CancellationException
        at java.util.concurrent.CompletableFuture.cancel(Unknown Source)
        at javacodegeeks.completionstageandcompletablefutureexample.CompletionStageAndCompletableFutureExam
val: 10.0
```

CompletionStageAndCompletableFuture

10. Download the Eclipse Project

This was an example about

```
CompletionStage
```

and

```
CompletableFuture
```

APIs from JDK8.

Download

You can download the full source code of this example here: **JavaCompletionStageAndCompletableFuture**

Tagged with: COMPLETIONSTAGE FUTURE

(No Ratings Yet) Start the discussion 1582 Views Tweet it!