Java 8: Writing asynchronous code with CompletableFuture

Last modified April 29, 2018

JAVA (/TAG/JAVA/)

Java 8 introduced a lot of cool features, whereas lambdas and streams caught much of the attention.

What you may have missed is the `CompletableFuture`.

You probably already know about Futures

A Future represents the pending result of an asynchronous computation. It offers a method — `get` — that returns the result of the computation when it's done.

The problem is that a call to `get` is blocking until the computation is done. This is quite restrictive and can quickly make the asynchronous computation pointless.

Sure — you can keep coding all scenarios into the job you're sending to the executor, but why should you have to worry about all the plumbing around the logic you really care about?

This is where `CompletableFuture` saves the day

Beside implementing the `Future` interface, `CompletableFuture` also implements the `CompletionStage` interface.

A `CompletionStage` is a promise. It promises that the computation eventually will be done.

The great thing about the `CompletionStage` is that it offers a vast selection of methods that let you attach callbacks that will be executed on completion.

This way we can build systems in a non-blocking fashion.

Ok, enough chatting, let's start coding!

The simplest asynchronous computation

Let's start with the absolute basics — creating a simple asynchronous computation.

```
CompletableFuture.supplyAsync(this::sendMsg);
```

It's as easy as that.

`supplyAsync` takes a `Supplier` containing the code we want to execute asynchronously — in our case the `sendMsg` method.

If you've worked a bit with Futures in the past, you may wonder where the `Executor` went. If you want to, you can still define it as a second argument. However, if you leave it out it will be submitted to the `ForkJoinPool.commonPool()`.

Attaching a callback

Our first asynchronous task is done. Let's add a callback to it!

The beauty of a callback is that we can say what should happen when an asynchronous computation is done without waiting around for the result.

In the first example, we simply sent a message asynchronously by executing `sendMsg` in its own thread.

Now let's add a callback where we notify about how the sending of the message went.

```
CompletableFuture.supplyAsync(this::sendMsg)
                 .thenAccept(this::notify);
```

`thenAccept` is one of many ways to add a callback. It takes a `Consumer` — in our case `notify` — which handles the result of the preceding computation when it's done.

Chaining multiple callbacks

If you want to continue passing values from one callback to another, `thenAccept` won't cut it since `Consumer` doesn't return anything.

To keep passing values, you can simply use `thenApply` instead.

`thenApply` takes a `Function` which accepts a value, but also return one.

To see how this works, let's extend our previous example by first finding a receiver.

```
    CompletableFuture.supplyAsync(this::findReceiver)
                     .thenApply(this::sendMsg)
                     .thenAccept(this::notify);
```

Now the asynchronous task will first find a receiver, then send a message to the receiver before it passes the result on to the last callback to notify.

Building asynchronous systems

When building bigger asynchronous systems, things work a bit differently.

You'll usually want to compose new pieces of code based on smaller pieces of code. Each of these pieces would typically be asynchronous — in our case returning `CompletionStage` s.

Until now, `sendMsg` has been a normal blocking function. Let's now assume that we got a `sendMsgAsync` method that returns a `CompletionStage` .

If we kept using `thenApply` to compose the example above, we would end up with nested `CompletionStage` s.

```
CompletableFuture.supplyAsync(this::findReceiver)
                 .thenApply(this::sendMsgAsync);

// Returns type CompletionStage<CompletionStage<String>>
```

We don't want that, so instead we can use `thenCompose` which allows us to give a `Function` that returns a `CompletionStage` . This will have a flattening effect like a flatMap.

```
CompletableFuture.supplyAsync(this::findReceiver)
                 .thenCompose(this::sendMsgAsync);

// Returns type CompletionStage<String>
```

This way we can keep composing new functions without losing the one layered `CompletionStage` .

Callback as a separate task using the *async suffix*

Until now all our callbacks have been executed on the same thread as their predecessor.

If you want to, you can submit the callback to the `ForkJoinPool.commonPool()` independently instead of using the same thread as the predecessor. This is done by using the *async suffix* version of the methods `CompletionStage` offers.

Let's say we want to send two messages at one go to the same receiver.

```
CompletableFuture<String> receiver
            = CompletableFuture.supplyAsync(this::findReceiver);

receiver.thenApply(this::sendMsg);
receiver.thenApply(this::sendOtherMsg);
```

In the example above, everything will be executed on the same thread. This results in the last message waiting for the first message to complete.

Now consider this code instead.

```
CompletableFuture<String> receiver
            = CompletableFuture.supplyAsync(this::findReceiver);

receiver.thenApplyAsync(this::sendMsg);
receiver.thenApplyAsync(this::sendMsg);
```

By using the *async suffix*, each message is submitted as separate tasks to the `ForkJoinPool.commonPool()`. This results in both the `sendMsg` callbacks being executed when the preceding calculation is done.

The key is — the asynchronous version can be convenient when you have several callbacks dependent on the same computation.

What to do when it all goes wrong

As you know, bad things can happen. And if you've worked with `Future` before, you know how bad it could get.

Luckily `CompletableFuture` has a nice way of handling this, using `exceptionally`.

```
CompletableFuture.supplyAsync(this::failingMsg)
                 .exceptionally(ex -> new Result(Status.FAILED))
                 .thenAccept(this::notify);
```

`exceptionally` gives us a chance to recover by taking an alternative function that will be executed if preceding calculation fails with an exception.

This way succeeding callbacks can continue with the alternative result as input.

If you need more flexibility, check out `whenComplete` and `handle` for more ways of handling errors.

Handling timeouts in a controlled way

Timeouts are something that us coders often need to take care of. Sometimes we simply can't hang around waiting for a computation to be done.

This also applies to CompletableFutures. We need a way to say to our CompletableFutures how long we're willing to wait and what to do if the time runs out.

This has been addressed in Java 9 by introducing two new methods that'll enable us to take care of timeouts — `orTimeout` and `completeOnTimeout`.

So say we got a hanging message, and we don't want to wait forever for it to finish.

What `orTimeout` does is to complete our CompletableFuture exceptionally if it doesn't complete within the timeout we specify.

```
CompletableFuture.supplyAsync(this::hangingMsg)
                 .orTimeout(1, TimeUnit.MINUTES);
```

There we go! If our hanging message doesn't complete in one minute, a `TimeoutException` will be thrown.

This exception can then be handled by the `exceptionally` callback we looked at earlier.

Another option is to use the `completeOnTimeout` which gives us the possibility to provide an alternative value.

To me this is way better than just throwing exceptions around, because it allows us to recover in a nice controlled manner.

```
CompletableFuture.supplyAsync(this::hangingMsg)
                 .completeOnTimeout(new Result(Status.TIMED_OUT),1, TimeUnit.MINUTES);
```

Now, if our hanging message doesn't return in time, we'll return a result with the status `TIMED_OUT`.

Callback depending on multiple computations

Sometimes it would be really helpful to be able to create a callback that is dependent on the result of two computations. This is where `thenCombine` becomes handy.

`thenCombine` allows us to register a `BiFunction` callback depending on the result of two `CompletionStage`s.

To see how this is done, let's in addition to finding a receiver also execute the heavy job of creating some content before sending a message.

```
CompletableFuture<String> to =
    CompletableFuture.supplyAsync(this::findReceiver);

CompletableFuture<String> text =
    CompletableFuture.supplyAsync(this::createContent);

to.thenCombine(text, this::sendMsg);
```

First, we've started two asynchronous jobs — finding a receiver and creating some content. Then we use `thenCombine` to say what we want to do with the result of these two computations by defining our `BiFunction`.

It's worth mentioning that there is another variant of `thenCombine` as well — called `runAfterBoth`. This version takes a `Runnable` not caring about the actual values of the preceding computation — only that they're actually done.

Callback dependent on one or the other

Ok, so we've now covered the scenario where you depend on two computations. Now, what about when you just need the result of one of them?

Let's say you have two sources of finding a receiver. You'll ask both, but will be happy with the first one returning with a result.

```
CompletableFuture<String> firstSource =
    CompletableFuture.supplyAsync(this::findByFirstSource);

CompletableFuture<String> secondSource =
    CompletableFuture.supplyAsync(this::findBySecondSource);

firstSource.acceptEither(secondSource, this::sendMsg);
```

As you can see, it's solved easily by `acceptEither` taking the two awaiting calculations and a `Consumer` that will be executed with the result of the first one to return.

Further information

That covers the basics of what `CompletableFuture` has to offer. There are still a few more methods to check out, so make sure to check out the documentation (https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/CompletableFuture.html) for more details.

Marius Herring
Passionate developer located in Oslo. Writing about software development here at Dead Code Rising.
(https://twitter.com/mariushe) (https://github.com/mariushe)
(https://www.facebook.com/deadcoderising)
(https://www.instagram.com/mariushe/)