|  |
| --- |
| Completion Stage  Async Comparison  Fork/Join v Executor Pool |

Abstract

There is an option on any of the Async -prefixed methods that belong to the CompletionStage *Interface* to use an Executor as an additional formal argument.

The CompletableFuture *Class* is one concrete implementor of the CompletionStage *Interface*.

****

Application Delivery

There is an option on any Github repository to acquire code using either a ZIP file, or the canonical git clone utility – (via HTTP or SSH):

A screenshot of a cell phone

Description automatically generated

Someone at this link: <https://bit.ly/2TAmWZn> ... suggested employing a license, or just using G-Drive and a ZIP file:

A picture containing bird, tree, flower

Description automatically generated

The trusty ZIP file approach seems the most secure, as there is no possibility of “contributing to the cause”.

Yes, you can enable branch restrictions (rejects unwanted pushes), as discussed here:

<https://bit.ly/2XvzqT6>

But the net effect of doing something in G-Drive is easier, faster and has no backdoor.

****

Application Goal

The application being demoed contrasts two basic options when adopting the Async variant of methods from the CompletionStage *Interface* – as implemented by the *Concrete Class* CompletableFuture.

In each case, no work is being done, just a comparison of the scaffolding that needs to be done in each option/use case.

Each approach has merit, if properly matched to the volume/volatility of their respective runtime environments.

There are no metrics relating to each approach.

An examination of the differences in the code for each approach is the only comparison being made.

****

Options

The application executes code for each approach/option.

Option One

Allow the Async method to default to the characteristics of an internal Fork/Join implementation, which typically employs 3 worker threads.

It is not clear whether the internal Fork/Join implementation remains available for extended periods, or whether at some interval, they are reclaimed, nor whether the internal Fork/Join implementation is shared.

Option Two

Adopt a strategy that allows the Async method to elect a custom Executor (Thread Pool) to replace the internal Fork/Join implementation.

This is accomplished by decorating the Async method signature with a reference to an already-constructed Executor (as a second formal argument).

The lifecycle of the custom Executor takes additional effort to shutdown properly (or, not).

This lifecycle management is inherent to any robust Executor -- as they can remain passive/quiescent, then be reused on-demand.

Remember, threads are an expensive resource.

****

Code Design

The application has two methods that are coupled together for continuity (there were 17 other such exploratory methods before).

The method-coupling (Lines 36-8) is an example of the ßuilder Design Pattern.

A screenshot of a computer screen

Description automatically generated

Method-coupling is enabled when the method(s) each return a reference to the (main thread’s) owning-thread’s class, via the keywords **return this**.

The participating methods must also have a return type that references their enclosing class.

****

Code, Option One (Fork/Join)

The default approach will, of course, have less code than a custom approach.

In the next screen shot, the Active Thread becomes the thread issued by the ForkJoinPool.

A screenshot of a computer screen

Description automatically generated

Above, the Active Thread reports on its state, both before and after the instance of CompletableFuture – having done no work – makes a call to its canonical join() method to signal it is done.

This signal indicates that the CompletableFuture (ForkJoinPool-based thread) relinquishes control to the main thread of control – namely, the application’s main thread.

Active Thread: ForkJoinPool.commonPool-worker-3

CompletableFuture join() is called

Active Thread Exits: main

Code, Option Two (Executor)

The custom approach will, of course, have more code than a default approach.

It is mainly the lifecycle code that contributes to the volume (not complexity) of the implementation.

For simplicity, the custom Executor (Thread Pool) code is configured/created inside the method (it could also originate in a factory, somewhere):

A screenshot of a computer screen

Description automatically generated

The CompletableFuture.runAsync() static factory method takes as a second argument (line 90) the instance of the custom Executor just instantiated:

A screenshot of a computer

Description automatically generated

One line 92, the CompletableFuture instance – having done no work – makes a call to its canonical join() method.

This call signals that the Thread that originated in the custom Executor instance relinquishes control to the main thread of control – namely, the application’s main thread.

If nothing else is done, the custom Executor instance will block, waiting for more work.

Since this is not likely what we want – in order to make a direct comparison with the ForkJoinPool-based thread approach – we will gracefully shut down the custom Executor instance.

Active Thread: FixedThreadPool-executor-1

CompletableFuture join() is called

Active Thread Exits: main

The initial shutdown logic – preceded by the (formerly active) Active Thread reporting on its state is – illustrate by the call on line 99 (previous page).

This call stops any further in-transit requests to the custom Executor and initiates the Executor shutdown sequence.

The balance of the shutdown lifecycle for the custom Executor is shown in the next screen shot (next page).

A screenshot of a computer

Description automatically generated

Having initiated a shutdown request, the custom Executor waits 30 seconds for any in-transit requests to finish (line 104), before making a more forceful shutdown request, thus forcing any residual requestors to terminate (line 107).

Having initiated a more forceful shutdown request, the custom Executor waits another 30 seconds for any in-transit requestors to respond to cancellation by finishing up, reporting a potential problem (line 110).

If anything goes really wrong, the catch block (lines 115-25) again demands a shutdown, then preserves the interrupt status of the current thread (line 123).

The precautionary interrupt call on the current thread on line 123: Facilitates the current thread's shutdown; Preserves the interrupt status of that thread; Reports that a task was in an incomplete/indeterminate state when it was interrupted.

****

Code, Running

A screenshot of a cell phone

Description automatically generated

****

Just ßad

The build logic reports a successful Maven build with a simple message: ßuild Ok (you are spared having to stare at Maven’s build progress):

A screenshot of a cell phone

Description automatically generated

However, when a Maven build FAIL occurs, the build logic reports the FAIL twice, with the Maven errata which was intercepted and cached shown in-between.

A screenshot of a cell phone

Description automatically generated

****

The ßuild Logic

The build logic looks like this:

A screenshot of a computer screen

Description automatically generated

Lines 12-18 Stop any applications running on port 8080. Absorb/discard any cruft from the kill is (line 17).

Line 21 Perform a standard Spring-boot build, but suppress annoying things like tests,

coloration, and console output – instead, cache the build’s output to a flat-file (for rainy-day reading).

Lines 24-33 Interrogate the outcome of the Maven build (status == 0 is a success). If not, send a message, and display the errata (the contents of the cached flat-file

MAVEN.LOG).

****

Run app w/o fanfare

A screenshot of a cell phone

Description automatically generated

****

Javadoc, CLI

There is always a CLI way to do whatever a GUI tool does (the GUI facility likely delegates to a CLI script anyway).

This section covers a straightforward, script’d way (jdoc) to generate Javadoc.

No GUI.

No Maven.

There are two modalities for the jdoc ßash script.

Report possible errors

Suppress all reporting

Usage: jdoc

The next screen shot shows the Javadoc script

****

Lines 4-10 Identify troublesome Spring artifacts for which we don’t want to generate

Javadoc, but have to use.

Lines 15-20 If the CLI command – a ßash script named: jdoc – accepted *any* argument,

debug/verbose reporting will be done.

Lines 22-27 If no argument was passed to the jdoc script, then you get pure functionality – just the Javadoc generation. This happens because all console output – the

OS’s output streams stdout/stderr – is redirected to the OS’s null device:

**> /dev/null 2>&1**

Line 31 Open a Javadoc HTML (reference) page in a browser

The jdoc ßash script deposits Javadoc

artifacts in the docs directory/folder,

in the application root

A screenshot of a computer screen

Description automatically generated