CompletableFuture

Azat Satklichov
azats@seznam.cz,
http://sahet.net/htm/java.html,
https://github.com/asatklichov/multithreading-java-vs-nodejs

CompletableFuture

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/CompletableFuture.html
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/class-use/CompletableFuture.html

Composing Futures - CompletableFuture(Java 8, 9) support non-blocking operations

Java v.1.8 (18-March 2014)
Java v.1.9 (21-September 2017)
newIncompleteFuture(), defaultExecutor(), delayedExecutor, failedFuture, …

Java v.12 (19-March 2019) Exception handling in CompletionStage – exceptionallyAsync (2x), exceptionallyComposeAsync,

Concepts behind the **CompletableFuture (uses ForkJoinPool by default** (designed for this purpose). **)** and **reactive programming** are reaching **parallelism** using **non-blocking operations**, It understands tasks depend on other tasks, so avoids blocking threads, and changing (context switch expensive) threads. It is an ideal candidate to write asynchronous systems.

In the case of asynchronous computation (it is usually multi steps), actions represented as callbacks tend to be either scattered across the code or deeply nested inside each other. Things get even worse when we need to handle errors that might occur during one of the steps.

Different ways to implement asynchronous programming in Java

1- Executor Service framework, Java Concurrency API (java.util.concurrent). In Core Thread API – on demand created thread dies when the task is completed, non reusable.

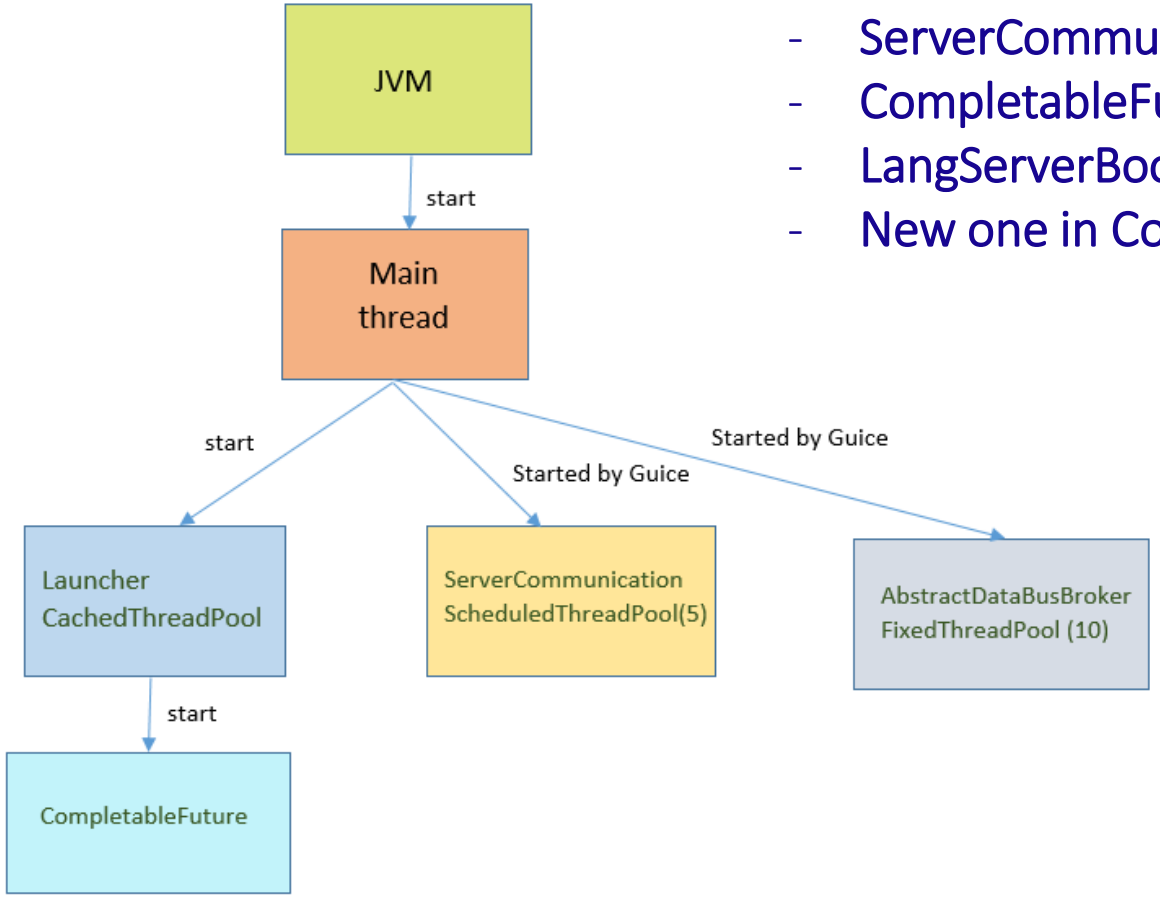
- ✓ **ExecutorService** – **async exec mechanism,** gives a task (Runnable, Callable), then gets a Future
- ✓ **Future** – represents result of an asynchronous computation when asynchronous task is created
- ✓ **Cached Thread Pool** – creating threads starting with 0 and max to 2^31-1. Task should not wait (**SynchronousQueue**) for execution. Limitation: system resources availability. Removes idle threads after 1-min. Good for a lot short-lived tasks.
- ✓ **Fixed Thread Pool** – keeps adding more tasks to the queue (**LinkedBlockingQueue**) in case all threads are busy. Good to control resource consumption, stack size, and tasks with unpredictable execution times.
- ✓ **Scheduled Thread Pool** - schedule commands to run after a given delay, or to execute periodically.
- ✓ **Single Thread Executor** - used for the purpose of executing tasks sequentially. For testing or using as event-loop.
- ✓ **ForkJoinPool** - ForkJoin provides parallel mechanism for CPU intensive tasks, uses work stealing alg. CompletableFuture, Parallel Streams uses it by default.
- ✓ **Guava** – via **MoreExecutors** class also daemon threads
- ✓ **Customizable Thread Pools** - to control .. resource consumption use **ThreadPoolExecutor** or **ScheduledThreadPoolExecutor**

LSP: `LangServerBootstrap(Launcher.Builder-newCachedThreadPool), AbstractDataBusBroker-fixed-pool(10), ServerCommunications(scheduled-pool(5), LangServerBootstrapTest (singleExecutor),`

Since Java 5, the **Future** represents the result of an async-computation but it **lacks** methods to chain/**combine** the computation steps **or handle possible** errors, can't be manually completed, and get() method is **blocked**.
You can not attach a callback function to the Future, and have it called automatically once Future's result is ready

2- Java 8, **CompletableFuture** class 3-@Async SpringBoot, etc…

Thread Pools in LSP Cobol



- LangServerBootstrap [Launcher](cached thread pool)
- AbstractDataBusBroker (fixed thread pool)
- ServerCommunications (scheduled thread pool)
- CompletableFuture (forkjoin common pool)
- LangServerBootstrapTest (single thread executor)
- New one in CobolTextDocumentService (cached thread pool)

- `MyThreadPoolExecutor (custom pool)`
- ServerCommunications (scheduled thread pool)
- CompletableFuture (forkjoin common pool) – providing manageable executor?

CompleteableFuture is like **PROMISES** in **JS. Node.js uses libuv**, whereas **web browser runtimes use browser-provided APIs** for the async I/O tasks. The event loop works pretty much the same in Node.js and in web browser JavaScript runtimes.
You can make a Promise and it's up to you to keep it ;)
When someone else makes you a promise you must wait to see if they honour it in the Future. CompletableFuture ;)

The Java 8 CompletableFuture and the **Guava SettableFuture (ListenableFuture)** can be thought of as **promises**, because their value can be set ("completed"), they also implement the Future interface, therefore there is no difference for the client.
The ***CompletableFuture*** implements both Future and *CompletionStage* interfaces, it can combine (pipeline) multiple asynchronous computations, handle possible errors and offers much more capabilities. **CompletionStage** lets you attach callbacks that will be executed on completion. CompeletableFuture is all about coordination ..
Around 70 methods for composing, combining, and executing asynchronous computation steps and handling errors.

Methods to chain CompletionStage (step of a chain, may trigger other task)

Task: What kind of task? Function(**apply**), Consumer(**accept**), Runnable(**run**)
- **3** types of methods, one for each type of task

Operation: What kind of operation does it support? 4 types
- Chaining (1:1)
- Composing (1:1)
- Combining, waiting for both result (2:1)
- Combining, triggered on the first available result (2:1)

e.g. Operation – you can chain, compose, or combine(&&, ||)

Thread: In which thread can be executed? **3** types
In the same executor as caller (main thread), In new executor, passed as parameter (executor), Asynchronously (ForkJoinPool)

1:1 patterns
- thenApply, thenApplyAsync, thenAccept, thenRunAsync, ..
- thenComposeAsync, ..

2:1 patterns (*Both*, *Combine* and *Either*)
- thenCombineAsync(CS, BiFunction), ..
- thenAcceptBoth(CS, BiConsumer), ..
- applyToEither(CS, Function), acceptEitherAsync, ..

So: 3*4*3=36 methods for chain, factory methods(?) exception related(?), complete(?), fail(?),delay(?), timeout(?)

Functional Interfaces

Functional interface	Predicate<T>	Consumer<T>
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<T, U>	(T, U) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

TASKS

Runnable () -> void, run() CF: ➔ thenRun()
Consumer T -> void accept() CF: ➔ thenAccept()

Callable () -> T, call()
Supplier () -> T, get() CF: ➔ supplyAsync

Function T -> R, apply(), CF ➔ thenApply()

`Callable` is more specialized than `Supplier`. If you are not dealing with another thread or your task is very unlikely to throw an exception, `Supplier` is recommended.

```
T join(); // may throw an unchecked exception
T getNow(T valueIfAbsent):

boolean complete(V value);
void obtrudeValue(V value);

boolean completeExceptionally(Throwable t);
void obtrudeException(Throwable t);
```

obtrudeValue(value):

Checks if the task is done

- if it is done: then forces the returned value to value

- if it is not, then it completes it and sets the returned value to value

This should be used in error recovery operations

CompletableFuture brings 5 future-like methods

- two methods to get the result in a different way

- two methods to force the returned value

- and two to force an exception

completeExceptionally(throwable):

- forces the completion if the task is not done

obtrudeException(throwable):

- forces the completion even if the task is done

Completions (as a Simple *Future or explicit):* complete, obtrudeValue, *get, join, getNow, completeExceptionally, ..*
***CompletableFuture* with Encapsulated Computation Logic:** runAsync(Runnable, ..), supplyAsync(Supplier, ..), completeAsync ***

Processing Results (attach callbacks): thenApply(Function), thenApply**Async**(), thenAccept(Consumer), thenRun(Runnable),
Manual completion: *completedFuture, completeExceptionally, obtrudeException*
cancel() is equivalent to completeExceptionally(new CancellationException()) , isCompletedExceptionally, idDone. It is not a bug, but a design decision.
Combining Futures(Consume PrevStageRes) thenCompose(Function), thenCombine(CS, ..), thenAcceptBoth(CS, .), acceptEither

(Monadic design pattern) Difference Between thenApply()[nested] **and** thenCompose(). *Analogous to … map* and *flatMap*

Running Multiple *Futures* in Parallel: allOf(CompletableFuture<?>... Cfs), anyOf, join

Handling Errors: exceptionally(e), handle(r,e). , whenComplete(Unlike handle) .

Async Methods – help to parallelize: ***, thenApply**Async**(), thenCompose**Async**() [.. ThreadPoolExecutor, shutdownNow()],
JDK 12 *CompletionStage new methods:* exceptionallyAsync, exceptionallyCompose, exceptionallyComposeAsync

JDK 9 *CompletableFuture* API: New factory methods added, Support for delays and timeouts. Improved support for subclassing. So API comes with 8 new instance methods: newIncompleteFuture()[**virtual constructor**], **copy**()[**defensive copying**], minimalCompletionStage(), completeAsync()-2x, defaultExecutor(), **orTimeout**(), completeOnTimeout() [**delay**] and 5 new static utility methods: delayedExecutor()-2x [**delay - schedule CF**], completedStage(), failedStage(), **`failedFuture`**()

In LSP Cobol: **`failedFuture(Java 9), completedFuture, complete, runAsync, supplyAsync,`** whenComplete**`, thenApply`**

References

https://medium.com/modern-mainframe/multithreading-in-java-vs-node-js-c558d59050c9
https://www.baeldung.com/java-completablefuture
https://github.com/vsilaev/tascalate-concurrent
https://dzone.com/articles/completablefuture-cant-be
https://www.nurkiewicz.com/2015/03/completablefuture-cant-be-interrupted.html
https://github.com/vsilaev/tascalate-concurrent
https://www.nurkiewicz.com/2014/05/interruptedexception-and-interrupting.html
https://www.jesperdj.com/2015/09/21/listen-to-the-future-with-google-guava/
https://dzone.com/articles/20-examples-of-using-javas-completablefuture
https://github.com/antlr/antlr4/pull/2610
https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html
https://codeahoy.com/java/How-To-Stop-Threads-Safely/

Source code:
https://github.com/asatklichov/multithreading-java-vs-nodejs

THANK YOU