TD MCP – session 6 – Future Composition

March 9, 2022

Learning objective: compose futures to avoid blocking as much as possible.

The first 5 exercises are essential.

Exercise 1: future composition prediction

What is the expected output of the following future composition?

```java
System.out.println("Launching computation on " + Thread.currentThread());
CompletableFuture.supplyAsync(() -> {
    System.out.println("Producing 5 on " + Thread.currentThread());
    Thread.sleep(1000);
    return 5;
})
    .thenApply(res -> {
        System.out.println("Adding 2 on " + Thread.currentThread());
        return res + 2;
    })
    .thenCombine(CompletableFuture.supplyAsync(() -> {
        System.out.println("Producing 8 on " + Thread.currentThread());
        return 8;
    }), (res1, res2) -> {
        var res = res1 * res2;
        System.out.println("Multiplying " + res + " on " +
            Thread.currentThread());
        return res;
    });
System.out.println("Finish launch on " + Thread.currentThread());
```

Launching computation on main
Producing 5 on thread-1
Producing 8 on thread-2
Finish launch on main

The order of the last three output lines is not guaranteed.
After 1000 milliseconds:

Adding 2 on thread-1
Multiplying 56 on thread-1

Note that this code is not particularly clear, in particular the fact that the second call to `supplyAsync` is done halfway in the code. We avoid this below.
Exercise 2: composition with sequence and join
Assume variables x, y and z are initialized, and functions f, g, h and F take some non-negligible amount of time to compute. Propose an asynchronous execution that avoids any blocking operation for the following computation: F(g(f(x) + y) + h(z)).

There are two independent execution flows (f and g for the first and h for the second) that join before executing the last function, F.

Even though it is possible to merge the processing done with thenApply, it reduces the number of tasks and may lead to fairness issue (head-of-line blocking). In contrast, too much tasks induces overhead. It is therefore advisable to keep a medium granularity (each task should perform a small but significant amount of work).

```java
CompletableFuture c1 = CompletableFuture.supplyAsync(() -> h(z));
CompletableFuture c2 = CompletableFuture.supplyAsync(() -> f(x))
    .thenApply((res) -> g(res + y))
    .thenCombine(c1, (res1, res2) -> F(res1 + res2));
```

Exercise 3: composition with condition
Similarly, propose an asynchronous execution that avoids any blocking operation for the following computation: first, we compute f(x); if the result is positive, we compute g(f(x)), otherwise, h(f(x)).

```java
CompletableFuture.supplyAsync(() -> f(x))
    .thenApply(res -> res > 0 ? g(res) : h(res))
```

Exercise 4: composition with fork
Similarly, propose an asynchronous execution that avoids any blocking operation for the following computation while minimizing the amount of computation: g(f(x)) + h(f(x)).

```java
CompletableFuture c1 = CompletableFuture.supplyAsync(() -> f(x));
CompletableFuture c2 = c1.thenApply(res -> h(res));
CompletableFuture c3 = c1.thenApply(res -> g(res));
CompletableFuture c4 = c2.thenCombine(c3, (x, y) -> x + y);
```

Alternatively, we can rely on thenCompose:

```java
CompletableFuture.supplyAsync(() -> f(x))
    .thenCompose(res -> {
        var cf = CompletableFuture.supplyAsync(() -> g(res));
        return CompletableFuture.supplyAsync(() -> h(res))
            .thenCombine(cf, (x, y) -> x + y)
    })
```

Alternatively, we can write:

```java
var fut1 = new CompletableFuture<Integer>();
var fut2 = new CompletableFuture<Integer>();
```
Exercise 5: rewriting allOf

We want to implement the behavior of allOf from class CompletableFuture that takes multiple completable futures as arguments. From a collection of runnables, build a completable future that represents their completion by relying on runAfterBoth.

With streams:

```java
runnables.stream()
    .map(CompletableFuture::supplyAsync)
    .reduce(((c1, c2) -> c1 CompletableFuture:runAfterBoth(c2, () -> {})))
    .get()
```

With loops:

```java
CompletableFuture<Void> fut = null;
for (Runnable runnable : runnables)
    if (fut == null)
        fut = CompletableFuture.supplyAsync(runnable)
    else
        fut = fut.runAfterBoth(CompletableFuture.supplyAsync(runnable), () -> {})
```

Exercise 6: rewriting anyOf

Same question with anyOf and runAfterEither.

With streams:

```java
runnables.stream()
    .map(CompletableFuture::supplyAsync)
    .reduce(((c1, c2) -> c1 CompletableFuture:runAfterEither(c2, () -> {})))
    .get()
```

With loops:

```java
CompletableFuture<Void> fut = null;
for (Runnable runnable : runnables)
    if (fut == null)
        fut = CompletableFuture.supplyAsync(runnable)
    else
        fut = fut.runAfterEither(CompletableFuture.supplyAsync(runnable), () -> {})
```