10: ♦ ExecutorService Vs Fork/Join & Future Vs CompletableFuture Interview Q&A

Posted on March 6, 2016 by Arulkumaran Kumaraswamipillai

Q1. What is the difference between "**ExecutorService**" and "**Fork/Join Framework**"?

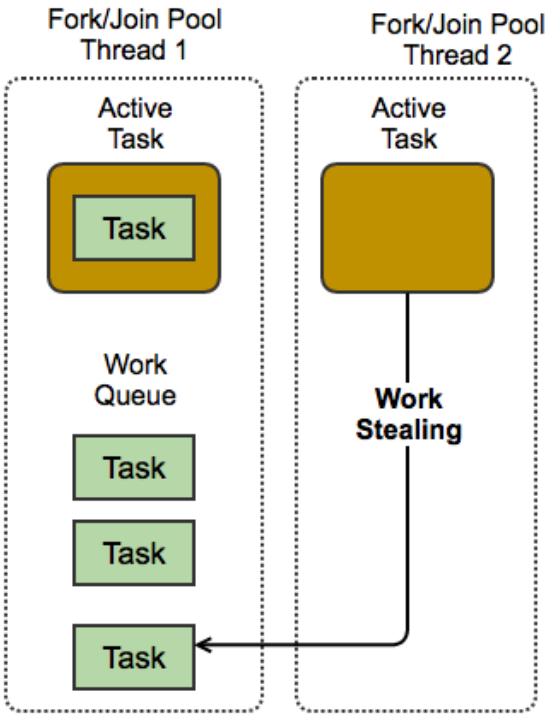
A1. The Fork/Join framework uses a special kind of thread pool known as the **ForkJoinPool**, which is a specialized implementation of **ExecutorService** implementing the **1) work-stealing** algorithm, in which the idle workers steal the work from those workers who are busy. This can give better performance.

9 tips to earn more | What can u do to go places? | **945+** paid members. LinkedIn Group. **Reviews**

600+ Full Stack Java/JEE Interview Q&As ♥Free ♦FAQs

open all | close all

- Ice Breaker Interview
- Core Java Interview C
  - Java Overview (4)
  - Data types (6)
  - constructors-metho
  - Reserved Key Wor
  - Classes (3)
  - Objects (8)
  - OOP (10)
  - GC (2)
  - Generics (5)
  - FP (8)
  - IO (7)



ForkJoinPool Work-Stealing

The Fork-Join breaks the task at hand into mini-tasks until the mini-task is simple enough that it can be solved without further breakups. The pseudo-code is like:

```
Result solve(Problem problem) {
    if (problem is small) //e.g. smaller than ba
        compute the result
    else {
        split problem into chunks
        fork new subtasks to solve each chunk
        join all subtasks
        compose result from subresults
    }
}
```

The actual working example can be found at Java multi-threading scenarios interview Q&A.

Another difference in ForkJoinPool compared to an ExecutorService is that, even though you specify an initial capacity, the

As a Java Architect

Java architecture & design concepts

2) ForkJoinPool adjusts its pool size dynamically in an attempt to maintain enough active threads at any given point in time.

3) ForkJoinPool threads are all daemon threads, hence its pool does not have to be explicitly shutdown as we do for executor service “executorService.shutdown()”;

Q2. What do you understand by the term “**asynchronous**” processing?
A2. **Asynchronous** task means its processing returns before the task is finished, and generally causes some work to happen in the background before triggering some **future** action in the application to get the results or handle the exception via one of the following mechanisms:

1) Callback argument
2) Return place holder. e.g. **Future**, Promise, etc.
3) Deliver to a queue.

Q3. Why was CompletableFuture class introduced in Java 8 to work with executor service when you already have “ForkJoinPool” that gives good performance?
A3. The use cases for the fork/join are pretty narrow. It can only be used when the following conditions are met.

1) The data-set is large & efficiently splittable.

2) The operations performed on individual data items should be reasonably independent of each other.

3) The operations should be expensive & CPU intensive.

When the above conditions do not hold, and your use case is more I/O or network intensive then use ExecutorService with **CompletableFuture**.

Q4. What is a Future interface, and why was it introduced in Java?
A4. The Future<V> represents the **result of an**

interview Q&As with diagrams | What should be a typical Java EE architecture?

Senior Java developers must have a good handle on

open all | close all
⊞ Best Practice (6)
⊞ Coding (26)
⊞ Concurrency (6)
⊞ Design Concepts (7)
⊞ Design Patterns (11)
⊞ Exception Handling (3
⊞ Java Debugging (21)
⊞ Judging Experience Ir
⊞ Low Latency (7)
⊞ Memory Managemen
⊞ Performance (13)
⊞ QoS (8)
⊞ Scalability (4)
⊞ SDLC (6)
⊞ Security (13)
⊞ Transaction Managen

80+ step by step Java Tutorials

open all | close all
⊞ Setting up Tutorial (6)
⊞ Tutorial - Diagnosis (2
⊞ Akka Tutorial (9)
⊞ Core Java Tutorials (2
⊞ Hadoop & Spark Tuto

asynchronous computation. The result is known as a future because the results will not be available until some moment in the future. You can invoke methods on the Future interface

1) Get the results by **blocking indefinitely** or for a **timeout to elapse** when the task hasn't finished.

2) Cancel a task.

3) Determine if a task has been cancelled or completed.

The 3 variants of submitting a task to the ExeccutorService returns a Future object.

```
1
2 Future submit(Callable task)
3 Future submit(Runnable task)
4 Future submit(Runnable task, T result)
5
```

In Java 8, the Runnable & Callable interfaces are annotated with "**@FunctionalInterface**". The major benefit of functional interface is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.

```
1
2  import java.util.concurrent.ExecutionException;
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.Future;
6  import java.util.function.Function;
7  import java.util.function.IntToDoubleFunction;
8
9  public class FutureExample {
10
11      static class Recursive<I> {
12          public I func;
13      }
14
15      static Function<Integer, Double> factorial =
16          Recursive<IntToDoubleFunction> recursive
17          recursive.func = n -> (n == 0) ? 1 : n *
18
19          return recursive.func.applyAsDouble(x);
20      };
21
22      public static void main(String[] args) throw
23          System.out.println(Thread.currentThread(
24
```

JEE Tutorials (19)
Scala Tutorials (1)
Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

100+ Preparing for pre-interview Java written home assignments & coding tests

open all | close all
Can you write code? (
♦ Complete the given
Converting from A to I
Designing your classe
Java Data Structures
Passing the unit tests
What is wrong with th
Writing Code Home A
Written Test Core Jav
Written Test JEE (1)

How good are your...to go places?

open all | close all
Career Making Know-
Job Hunting & Resum

```
25          ExecutorService newFixedThreadPool = Exe
26
27          final int nthFactorial = 25;
28
29          Future<Double> result = newFixedThreadPo
30              System.out.println(Thread.currentThr
31                  Double factorialResult = factori
32                  return factorialResult;
33              });
34
35          System.out.println("isDone = " + result.
36          System.out.println("isCancelled = " + re
37
38          // result.cancel(true); //You may cancel
39
40          Double res = result.get(); // 3 blocked
41
42          System.out.println(Thread.currentThread(
43
44          newFixedThreadPool.shutdown();
45      }
46  }
47
```

Outputs:

```
1
2  main thread enters main method
3  pool-1-thread-1 factorial task is called
4  isDone = false
5  isCancelled = false
6  main result is 1.551121004333098 6E25
7
```

Note: The “factorial” static function is explained at Top 6 tips to transforming your thinking from OOP to FP with examples

As you can see, there is a **main thread**, which spawns a **new thread** to **asynchronously** find out the 25th factorial. The factorial function is a time consuming function, and then main thread continues without being blocked whilst the factorial task is being executed in the background. The main thread gets blocked when it reaches the “**get()**” method call on the Future object “result” at line “//3”. Finally, the main thread prints the result of 25th factorial.

Q5. Why was **CompletableFuture** introduced in Java 8 when you already had the Future interface?
A5. The CompletableFuture implement both **CompletionStage**<<T> and **Future**<T> interfaces. Hence it provides the functionality of the Future interface in terms of

getting the result, canceling a task, and determining if a task is completed or cancelled, etc. Since it also implements the "CompletionStage" interface it provides **1)** functionality to **join & combine** CompletableFuture objects and **recover** from exceptional scenarios.

2) The "**Future.get()**" is a blocking call. Whereas, given a CompletableFuture "f" executing a task, you can both synchronously and asynchronously run another task as a **callback** upon completion of "f".

If you need a result:

```
1
2 f.thenApply(result -> isDone(result));      // sy
3 f.thenApplyAsync(result -> isDone(result)); // as
4
```

If you don't need a result:

```
1
2 f.thenRun(() -> isDone());            // sync cal
3 f.thenRunAsync(() -> isDone());       // async ca
4
```

The same logic as above for "FutureExample" using a **CompletableFuture** object:

```
 1
 2 import java.util.concurrent.CompletableFuture;
 3 import java.util.concurrent.ExecutionException;
 4 import java.util.concurrent.ExecutorService;
 5 import java.util.concurrent.Executors;
 6 import java.util.function.Function;
 7 import java.util.function.IntToDoubleFunction;
 8
 9 public class FutureExample {
10
11     static class Recursive<I> {
12         public I func;
13     }
14
15     static Function<Integer, Double> factorial =
16         Recursive<IntToDoubleFunction> recursive
17         recursive.func = n -> (n == 0) ? 1 : n *
18         return recursive.func.applyAsDouble(x);
19     };
20
21     public static void main(String[] args) throw
```

```
22          System.out.println(Thread.currentThread(
23
24          ExecutorService newFixedThreadPool = Exe
25
26          final int nthFactorial = 25;
27
28          CompletableFuture<Double> result = Compl
29              System.out.println(Thread.currentThr
30                  Double factorialResult = factori
31                  return factorialResult;
32              }, newFixedThreadPool);
33
34          System.out.println("isDone = " + result.
35          System.out.println("isCancelled = " + re
36
37          // result.cancel(true); //You may cancel
38
39          Double res = result.get(); // blocked un
40
41          System.out.println(Thread.currentThread(
42
43          newFixedThreadPool.shutdown();
44      }
45
46  }
47
```

Chaining CompletionFuture Example

As shown below, new CompletionFuture objects can be created from another and also chained as demonstrated below at //1, //2, //3, and //4.

```
1
2  import java.util.concurrent.CompletableFuture;
3  import java.util.concurrent.ExecutionException;
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.Executors;
6  import java.util.function.Function;
7  import java.util.function.IntToDoubleFunction;
8
9  public class FutureExample {
10
11      // recursion declaration
12      static class Recursive<I> {
13          public I func;
14      }
15
16      static Function<Integer, Double> factorial =
17          Recursive<IntToDoubleFunction> recursive
18          recursive.func = n -> (n == 0) ? 1 : n *
19
20          return recursive.func.applyAsDouble(x);
21      };
22
23      static Function<Double, Double> square = x -
24          return x * x;
25      };
26
```

```
27      public static void main(String[] args) throw
28          System.out.println(Thread.currentThread(
29
30          ExecutorService newFixedThreadPool = Exe
31
32          final int nthFactorial = 5;
33
34          // 1 CompletableFuture stage for calcula
35          CompletableFuture<Double> factorialCalcS
36              System.out.println(Thread.currentThr
37              Double factorialResult = factorial.a
38              return factorialResult;
39          }, newFixedThreadPool);
40
41          // 2 creates another CompletableFuture.
42          factorialCalcStage.thenAcceptAsync(t ->
43
44          // 3 creates another CompletableFuture f
45          CompletableFuture<Double> squareCalcStag
46              System.out.println(Thread.currentThr
47              Double squareResult = square.apply(r
48
49              return squareResult;
50          }, newFixedThreadPool);
51
52          // 4 creates another CompletableFuture.
53          CompletableFuture<Void> lastStage = squa
54                  .println(Thread.currentThread().
55
56          lastStage.join(); //waits for the lastSt
57
58          newFixedThreadPool.shutdown();
59
60      }
61  }
62
```

Outputs:

```
1
2  main thread enters main method
3  pool-1-thread-1 factorial task is called
4  pool-1-thread-2 square task is called
5  ForkJoinPool.commonPool-worker-2 The result after
6  ForkJoinPool.commonPool-worker-1 The result after
7
```

Q6. What is the difference between the calls thenAccept**Async**(…) and thenAccept(….)?

A6. In the above example, the thenAccept**Async**(…) are executed in a separate default thread pool known as the “ForkJoinPool.commonPool” and the order in which the lines “The result after square:” and “result after factorial: 120.0” vary between the runs. If you want to execute these two calls immediately after “pool-1-thread-1 factorial task is called” and “pool-1-thread-2 square task is called” in the same thread

then use **thenAccept**(….), in the lines **\\2** & **\\4** which will give an output as shown below:

The output will be:

```
1
2 main thread enters main method
3 pool-1-thread-1 factorial task is called
4 pool-1-thread-2 square task is called
5 pool-1-thread-2 The result after square: 14400.0
6 pool-1-thread-1 The result after factorial: 120.0
7
```

Now, if you want to run the “square task” synchronously in the same thread as “factorial task” then you need to make the following 2 changes as as shown to line //3.

1) thenApplyAsync(.., ..) to thenApply(..);

2) Remove the second argument “newFixedThreadPool” as thenApply(..) only takes a task.

```
1
2           // 3 creates another CompletableFuture f
3           CompletableFuture<Double> squareCalcStag
4               System.out.println(Thread.currentThr
5               Double squareResult = square.apply(r
6
7               return squareResult;
8           });
9
10
```

The output will be:

```
1
2 main thread enters main method
3 pool-1-thread-1 factorial task is called
4 pool-1-thread-1 square task is called
5 pool-1-thread-1 The result after square: 14400.0
6 pool-1-thread-1 The result after factorial: 120.0
7
```

Q7. What is a **task** from the above examples that is passed to a CompletableFuture object?

A7. A task can be

1) A **Runnable**: that takes no arguments and returns no arguments. “CompletableFuture.**runAsync**(runnable)”

```
1
2 CompletableFuture<Void> cfRunnableExample = Compl
3
4     @Override
5     public void run() {
6        //do something
7     }
8  });
9
```

2) A **Consumer**: that takes an object as an argument, but returns nothing.
“CompletableFuture.**thenAcceptAsync**(**Consumer**)”.

```
1
2 factorialCalcStage.thenAcceptAsync(t -> System.ou
3
```

3) A **Function**: That takes an object argument and returns an object result. “CompletableFuture.**supplyAsync**(**Supplier**)”. “Supplier” is a function that returns a value.

```
1
2  CompletableFuture<Double> factorialCalcStage = C
3                  .supplyAsync(() -> {
4                      System.out.println(Thread.cu
5                              + " factorial task i
6                          Double factorialResult =
7                          return factorialResult;
8                      }, newFixedThreadPool);
9
10
```

Q8. The CompletableFuture takes a task and optionally an executor service as arguments. What happens only if a task is supplied without an executor service?
A8. By default supplyAsync() uses **ForkJoinPool.commonPool()** thread pool shared between all CompletableFutures. This implicitly is a hard-coded thread pool that is completely outside of your control. So, you should always specify your own ExecutorService as shown in the above examples as in “**newFixedThreadPool**“.

Q9. What do you understand by the term **reactive programming** (RP)?
A9. The text book definition is that:

"Reactive programming is programming with asynchronous data streams."

As we saw earlier in the *CompletableFuture* methods like supplyAsync(…), thenApplyAsync(….), thenCombineAsync(….), thenApply(), etc where you can join (i.e. chain) and combine CompletableFuture stages to build a pipeline of later running tasks. These tasks can run asynchronously and provide synchronization methods to join or combine split parallel tasks. This helps you to parallelize your code to make an application more **reactive** and responsive.

Reactive programming example

1) Lines //1 & //2 are long running asynchronous tasks that run in parallel. Long run is simulated with 5 seconds sleep.

2) Line //3 is executed when //1 & //2 are completed.

```
1
2  import java.util.ArrayList;
3  import java.util.Arrays;
4  import java.util.concurrent.CompletableFuture;
5  import java.util.concurrent.TimeUnit;
6
7  public class ReactiveProgramming {
8
9      public static void main(String[] args) {
10
11         // 1. long running asynchronous task
12         final CompletableFuture<ArrayList<String
13             // lookup all students ... this can
14                 sleep(5);
15                 return new ArrayList<String>(Arr
16             });
17
18         // 2. long running asynchronous task
19         final CompletableFuture<ArrayList<String
20             // lookup all subjects ... this can
21                 sleep(5);
22                 return new ArrayList<String>(Arr
23             });
```

```
24
25          //3 combine //1 & //2 to produce a repor
26          final CompletableFuture<String> report =
27              System.out.println("Thread " + Threa
28              return u.toString() + s.toString();
29          });
30
31          System.out.println(report.join()); // re
32      }
33
34      private static void sleep(long seconds) {
35          try {
36              System.out.println("Thread " + Threa
37              TimeUnit.SECONDS.sleep(seconds);
38          } catch (InterruptedException e) {
39              e.printStackTrace();
40          }
41      }
42  }
43
```

Outputs:

```
1
2  Thread ForkJoinPool.commonPool-worker-1 is proces
3  Thread ForkJoinPool.commonPool-worker-2 is proces
4  Thread ForkJoinPool.commonPool-worker-2 is produc
5  [John, Peter, Sam][English, Maths, Science]
6
7
```

Home Assignment on Functional Reactive Programming

Create a simple framework where work items can be submitted using FP & CompletableFuture objects.

Popular Posts

♦ 11 Spring boot interview questions & answers

893 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

851 views

18 Java scenarios based interview Questions and Answers

456 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

413 views

♦ 7 Java debugging interview questions & answers

316 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

303 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

301 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

290 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

259 views

8 Git Source control system interview questions & answers

216 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job

interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

‹ 1. ♥ Set up environment for home coding assignments

3. Multi-Threading – Create a simple framework where work items can be submitted ›

Posted in Java 8, Multithreading, Reactive Programming

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category ▼

© Disclaimer

The contents in this Java-Success are copy righted. The author has the right to correct or enhance the current content without any prior notice.

These are general advice only, and one needs to take his/her own circumstances into consideration. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. No guarantees are made regarding the accuracy or usefulness of content, though I do make an effort to be accurate. Links to external sites do not imply endorsement of the linked-to sites.

© 2016 Java-Success.com ↑ Responsive Theme powered by WordPress