Advanced Java CompletableFuture Features: Factory Method Internals

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how factory methods initiate async computations
- Know how to apply factory methods
- Appreciate the internals of factory methods
  - Show how supplyAsync() maps to the Common Fork-Join Pool

```
CompletableFuture<BigFraction>
future = CompletableFuture
   .supplyAsync(() -> {
         BigFraction bf1 =
           new BigFraction(f1);
         BigFraction bf2 =
           new BigFraction(f2);

         return bf1
           .multiply(bf2);});
```

Learning Objectives in this Part of the Lesson

- Understand how factory methods initiate async computations
- Know how to apply factory methods
- Appreciate the internals of factory methods
  - Show how supplyAsync() maps to the Common Fork-Join Pool
  - See how supplyAsync() runs a supplier lambda concurrently & asynchronously

Mapping supplyAsync() to the Common Fork-Join Pool

Mapping supplyAsync() to the Common Fork-Join Pool

- supplyAsync() arranges to run the supplier lambda param concurrently & asynchronously in a thread residing in the Java common fork-join pool

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

Mapping supplyAsync() to the Common Fork-Join Pool

- supplyAsync() arranges to run the supplier lambda param concurrently & asynchronously in a thread residing in the Java common fork-join pool

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

supplyAsync() does not create a new thread!

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#supplyAsync

Mapping supplyAsync() to the Common Fork-Join Pool

- supplyAsync() arranges to run the supplier lambda param concurrently & asynchronously in a thread residing in the Java common fork-join pool

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

Instead, it return a future that's completed by a worker thread running in common fork-join pool

See dzone.com/articles/be-aware-of-forkjoinpoolcommonpool

Mapping supplyAsync() to the Common Fork-Join Pool

- supplyAsync() arranges to run the supplier lambda param concurrently & asynchronously in a thread residing in the Java common fork-join pool

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

supplyAsync()'s param is a supplier lambda that multiplies two BigFraction objects

Mapping supplyAsync() to the Common Fork-Join Pool

- supplyAsync() arranges to run the supplier lambda param concurrently & asynchronously in a thread residing in the Java common fork-join pool

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
        BigFraction bf1 =
          new BigFraction(f1);
        BigFraction bf2 =
          new BigFraction(f2);

        return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

Although Supplier.get() takes no params, effectively final values can be passed to this supplier lambda

See javarevisited.blogspot.com/2015/03/what-is-effectively-final-variable-of.html

Mapping supplyAsync() to the Common Fork-Join Pool

- supplyAsync() arranges to run the supplier lambda param concurrently & asynchronously in a thread residing in the Java common fork-join pool

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
      BigFraction bf1 =
        new BigFraction(f1);
      BigFraction bf2 =
        new BigFraction(f2);

      return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

The worker thread calls the Supplier.get() method to obtain this supplier lambda & perform the computation

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#supplyAsync

Internals of Completable Future Factory Methods

Internals of Completable Future Factory Methods

- supplyAsync() is implemented by leveraging a message-passing framework that feeds tasks to the Java common fork-join pool

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

Here's how supplyAsync() code uses the supplier passed to it

See classes/java/util/concurrent/CompletableFuture.java

Internals of Completable Future Factory Methods

- supplyAsync() is implemented by leveraging a message-passing framework that feeds tasks to the Java common fork-join pool

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

```
() -> { ... return
  bf1.multiply(bf2);
}
```

The supplier parameter is bound to the lambda passed to supplyAsync()

Internals of Completable Future Factory Methods

- supplyAsync() is implemented by leveraging a message-passing framework that feeds tasks to the Java common fork-join pool

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

Create an "incomplete" future here that's just a placeholder

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#CompletableFuture

Internals of Completable Future Factory Methods

- supplyAsync() is implemented by leveraging a message-passing framework that feeds tasks to the Java common fork-join pool

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

The supplier & incomplete future are encapsulated in an AsyncSupply message

Internals of Completable Future Factory Methods

- supplyAsync() is implemented by leveraging a message-passing framework that feeds tasks to the Java common fork-join pool

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

This message is enqueued for async execution in common fork-join pool.

This is an example of "message-driven" design *a la* Reactive programming!

Internals of Completable Future Factory Methods

- supplyAsync() is implemented by leveraging a message-passing framework that feeds tasks to the Java common fork-join pool

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

The incomplete future is returned to the caller for subsequent use (e.g., with completion stage methods)

Internals of Completable Future Factory Methods

- AsyncSupply is a nested class that executes the supplier lambda param in a thread residing in the Java common fork-join pool

```
static final class AsyncSupply<U> extends Async {
  final Supplier<U> fn;
  final CompletableFuture<U> dst;

  AsyncSupply(Supplier<U> fn, CompletableFuture<T> dst)
  { this.fn = fn; this.dst = dst; }

  public final boolean exec() {
    ...
    U u = fn.get();
    ...
    d.internalComplete(u, ex);
    ...
```

See classes/java/util/concurrent/CompletableFuture.java

Internals of Completable Future Factory Methods

- AsyncSupply is a nested class that executes the supplier lambda param in a thread residing in the Java common fork-join pool

```
static final class AsyncSupply<U> extends Async {
  final Supplier<U> fn;
  final CompletableFuture<U> dst;

  AsyncSupply(Supplier<U> fn, CompletableFuture<T> dst)
  { this.fn = fn; this.dst = dst; }

  public final boolean exec() {
    ...
    U u = fn.get();
    ...
    d.internalComplete(u, ex);
    ...
```

Async extends ForkJoinTask & Runnable so it can be executed

See classes/java/util/concurrent/CompletableFuture.java

Internals of Completable Future Factory Methods

- AsyncSupply is a nested class that executes the supplier lambda param in a thread residing in the Java common fork-join pool

```
static final class AsyncSupply<U> extends Async {
  final Supplier<U> fn;
  final CompletableFuture<U> dst;

  AsyncSupply(Supplier<U> fn, CompletableFuture<T> dst)
  { this.fn = fn; this.dst = dst; }

  public final boolean exec() {
    ...
    U u = fn.get();
    ...
    d.internalComplete(u, ex);
    ...
```

```
() -> { ... return
  bf1.multiply(bf2); }
```

AsyncSupply stores the original supplier lambda passed into supplyAsync()

Internals of Completable Future Factory Methods

- AsyncSupply is a nested class that executes the supplier lambda param in a thread residing in the Java common fork-join pool

```
static final class AsyncSupply<U> extends Async {
  final Supplier<U> fn;
  final CompletableFuture<U> dst;

  AsyncSupply(Supplier<U> fn, CompletableFuture<T> dst)
  { this.fn = fn; this.dst = dst; }

  public final boolean exec() {
    ...
    U u = fn.get();
    ...
    d.internalComplete(u, ex);
    ...
```

```
() -> { ... return
  bf1.multiply(bf2);
}
```

A worker thread then runs the supplier lambda asynchronously & stores the result

Internals of Completable Future Factory Methods

- AsyncSupply is a nested class that executes the supplier lambda param in a thread residing in the Java common fork-join pool

```
static final class AsyncSupply<U> extends Async {
  final Supplier<U> fn;
  final CompletableFuture<U> dst;

  AsyncSupply(Supplier<U> fn, CompletableFuture<T> dst)
  { this.fn = fn; this.dst = dst; }

  public final boolean exec() {
    ...
    U u = fn.get();
    ...
    d.internalComplete(u, ex);
    ...
```

get() can use the ForkJoinPool Managed Blocker mechanism to auto-scale the common pool size for blocking operations

See earlier lesson on "*The Java Fork-Join Pool: the ManagedBlocker Interface*"

Internals of Completable Future Factory Methods

- AsyncSupply is a nested class that executes the supplier lambda param in a thread residing in the Java common fork-join pool

```
static final class AsyncSupply<U> extends Async {
  final Supplier<U> fn;
  final CompletableFuture<U> dst;

  AsyncSupply(Supplier<U> fn, CompletableFuture<T> dst)
  { this.fn = fn; this.dst = dst; }

  public final boolean exec() {
    ...
    U u = fn.get();
    ...
    d.internalComplete(u, ex);
    ...
```

Trigger completion of the future using the encoding of the given arguments

End of Advanced Java CompletableFuture Features: Factory Method Internals