The Fork/Join Framework in Java 7

BY

Kishori Sharan

January 20, 2012

Code examples in this post are from *Chapter 7. Threads* of the book Harnessing Java 7 (Volume 2)

Java 5 and Java 6 added several concurrency constructs to the `java.util.concurrent` package to help developers deal with problems involving advanced level of concurrency. The executor framework was one of the several great additions in Java 5. Java 7 added a new implementation of the executor service, which is known as a fork/join framework. The focus of the fork/join framework is to solve those problems efficiently, which may use the divide-and-conquer algorithm, by taking advantage of the multiple processors on a machine. Before Java 7, we had several Java constructs to help us solve the problems that involved concurrency. The fork/join framework helps us solve the problems that involve parallelism. Typically, the fork/join framework is suitable in a situation where:

- A task can be divided in multiple subtasks that can be executed in parallel.
- When subtasks are finished, the partial results can be combined to get the final result.

The fork/join framework creates a pool of threads to execute the subtasks. When a thread is waiting on a subtask to finish, the fork/join framework uses that thread to execute other pending subtasks of other threads. The technique of an idle thread executing other thread's task is called *work stealing*. The fork/join framework uses a work-stealing algorithm to enhance the performance.

The following four classes are central to learning the fork/join framework. All of them are in the `java.util.concurrent` package.

- `ForkJoinPool`
- `ForkJoinTask`
- `RecursiveAction`
- `RecursiveTask`

An instance of the `ForkJoinPool` class represents a thread pool. An instance of the `ForkJoinTask` class represents a task. The `ForkJoinTask` class is an `abstract` class. It has two concrete subclasses: `RecursiveAction` and `RecursiveTask`. The framework supports two types of tasks: a task that does not yield a result and a task that yields a result. An instance of the `RecursiveAction` class represents a task that does not yield a result. An instance of the `RecursiveTask` class represents a task that yields a result.

Both classes, `RecursiveAction` and `RecursiveTask`, provide an abstract `compute()` method. Your class whose object represents a fork/join task should inherit from one of these classes and provide an implementation for the `compute()` method. Typically, the logic inside the `compute()` method is written on the following pattern.

```
if (Task is small) {
     Solve the task directly.
}
else {
     Divide the task into subtsaks.
     Launch the subtasks asynchronously (the fork stage).
     Wait for the subtasks to finish (the join stage).
     Combine the results of all subtasks.
}
```

The following two methods of the `ForkJoinTask` class provide two important features during a task execution.

- The `fork()` method launches a new subtask from a task for an asynchronous execution.
- The `join()` method lets a task wait for another task to complete.

Using the fork/join framework involves the following five steps.

Step-1: Declaring a class to represent a task

Create a class inheriting from the `RecursiveAction` or `RecursiveTask` class. An instance of this class represents a task that you want to execute. If your task yields a result, you need to inherit it from the `RecursiveTask` class. Otherwise, you would inherit it from the `RecursiveAction` class. The `RecursiveTask` is a generic class. It takes a type parameter, which is the type of the result of your task. A `MyTask` class that returns a `Long` result may be declared as follows.

```
public class MyTask extends RecursiveTask<Long> {
     /* Code for your task goes here */
}
```

Step-2: Implementing the compute() method

The logic to execute your task goes inside the `compute()` method of your class. The return type of the `compute()` method is the same as the type of the result that your task returns. The declaration for the `compute()` method of the `MyTask` class would look as shown below.

```
public class MyTask extends RecursiveTask<Long> {
     public Long compute() {
          /* Logic for the task goes here */
     }
}
```

Step-3: Creating a fork/join thread pool

You can create a pool of worker threads to execute your task using the `ForkJoinPool` class. The default constructor of this class creates a thread of pool, which has the same parallelism as the number of processors available on the machine.

```
ForkJoinPool pool = new ForkJoinPool();
```

Other constructors let you specify the parallelism and other properties of the pool.

Step-4: Creating the fork/join task

You need to create an instance of your task.

```
MyTask task = MyTask();
```

Step-5: Submit the task to the fork/join pool for execution

You need to call the `invoke()` method of the `ForkJoinPool` class passing your task as an argument. The `invoke()` method will return the result of the task, if your task returns a result. The following statement will execute our task.

```
long result = pool.invoke(task);
```

Let us consider a simple example of using the fork/join framework. We will generate a few random integers and compute their sum. Listing 1 has the complete code for our task. The class is named `RandomIntSum`. It extends `RecursiveTask<Long>`, because it yields a result of `Long` type. The result is the sum of all random integers. It declares a `randGenerator` instance variable, which is used to generate a random number. The `count` instance variable stores the number of random numbers that we want to use. The value for the `count` instance variable is set in the constructor.

The `getRandomInteger()` method of the `RandomIntSum` class generates a random integer between 1 and 100, prints a message with the integer value on the standard output, and returns the random integer.

The `compute()` method contains the main logic to perform the task. If the number of random numbers to use is 1, it computes the result and returns it to the caller. If the number of random number is more than one, it launches as many subtasks as the number of random numbers. Note that if we use ten random numbers, it will launch ten subtasks, because each random number can be computed independently. We will need to combine the results from all subtasks. Therefore, we need to keep the references of the subtask for later use. We use a `List` to store the references of all subtasks. Note the use of the `fork()` method to launch a subtask. The following snippet of code performs this logic.

```
List<RecursiveTask<Long>> forks = new ArrayList<>();
for(int i = 0; i < this.count; i++) {
     RandomIntSum subTask = new RandomIntSum(1);
     subTask.fork();

     /* Keep the subTask references to combine the results at the end */
     forks.add(subTask);
}
```

Once all subtasks are launched, we need to wait on all subtasks to finish and combine all random numbers to get the sum. The following snippet of code performs this logic. Note the use of the `join()` method, which will make the current task wait on the subtask to finish.

```
for(RecursiveTask<Long> subTask : forks) {
     result = result + subTask.join();
}
```

Finally, the `compute()` method returns the result, which is the sum of all the random numbers. Listing 2 has the code to execute a task, which is an instance of the `RandomIntSum` class.

This is a very simple example of using the fork/join framework. You are advised to explore the fork/join framework classes to know more about the framework. Inside the `compute()` method of your task, you can have complex logic to divide tasks into subtasks. Unlike in our example, you may not know in advance how many subtasks you need to launch. You may launch a subtask, which may launch another subtask and so on.

Listing 1: A ForkJoinTask class to compute the sum of a few random integers

```
// RandomIntSum.java
package com.jdojo.chapter7;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.RecursiveTask;

public class RandomIntSum extends RecursiveTask<Long> {
     private static Random randGenerator = new Random();
     private int count;

     public RandomIntSum(int count) {
          this.count = count;
     }
```

```
    @Override
    protected Long compute() {
         long result = 0;

         if (this.count <= 0) {
              return 0L; /* We do not have anything to do */
         }

         if (this.count == 1) {
              /* Compute the number directly and return the result */
              return (long) this.getRandomInteger();
         }

         /* Multiple numbers. Divide them into many single tasks. Keep
            the references of all tasks to call join() method later */
         List<RecursiveTask<Long>> forks = new ArrayList<>();

         for(int i = 0; i < this.count; i++) {
              RandomIntSum subTask = new RandomIntSum(1);
              subTask.fork();

              /* Keep the subTask references to combine the results
                 later */
              forks.add(subTask);
         }

         /* Now wait for all subtasks to finish and combine the result */
         for(RecursiveTask<Long> subTask : forks) {
              result = result + subTask.join();
         }

         return result;
    }

    public int getRandomInteger() {
         /* Generate the next randon integer between 1 and 100 */
         int n = randGenerator.nextInt(100) + 1;

         System.out.println("Generated a random integer: " + n);

         return n;
    }
}
```

Listing 2: Using a fork/join pool to execute a fork/join task

```
// ForkJoinTest.java
package com.jdojo.chapter7;
```

```
import java.util.concurrent.ForkJoinPool;

public class ForkJoinTest {
     public static void main(String[] args) {
          /* Create a ForkJoinPool to run the task */
          ForkJoinPool pool = new ForkJoinPool();

          /* Create an instance of the task */
          RandomIntSum task = new RandomIntSum(3);

          /* Run the task */
          long sum = pool.invoke(task);

          System.out.println("Sum is " + sum);
     }
}
```

Output: (You may get a different output.)

```
Generated a random integer: 62
Generated a random integer: 46
Generated a random integer: 90
Sum is 198
```