(http://baeldung.com)

Guide to the Fork/Join Framework in Java

Last modified: July 20, 2017

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

The fork/join framework was presented in Java 7. It provides tools to help speed up parallel processing by attempting to use all available processor cores – which is accomplished **through a divide and conquer approach**.

In practice, this means that **the framework first "forks"**, recursively breaking the task into smaller independent subtasks until they are simple enough to be executed asynchronously.

After that, **the "join" part begins**, in which results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed.

To provide effective parallel execution, the fork/join framework uses a pool of threads called the *ForkJoinPool*, which manages worker threads of type *ForkJoinWorkerThread*.

2. *ForkJoinPool*

The *ForkJoinPool* is the heart of the framework. It is an implementation of the *ExecutorService (/java-executor-service-tutorial)* that manages worker threads and provides us with tools to get information about the thread pool state and performance.

Worker threads can execute only one task at the time, but the *ForkJoinPool* doesn't create a separate thread for every single subtask. Instead, each thread in the pool has its own double-ended queue (or deque (https://en.wikipedia.org/wiki/Double-ended_queue), pronounced *deck*) which stores tasks.

This architecture is vital for balancing the thread's workload with the help of the **work-stealing algorithm.**

2.1. Work Stealing Algorithm

Simply put – free threads try to "steal" work from deques of busy threads.

By default, a worker thread gets tasks from the head of its own deque. When it is empty, the thread takes a task from the tail of the deque of another busy thread or from the global entry queue, since this is where the biggest pieces of work are likely to be located.

This approach minimizes the possibility that threads will compete for tasks. It also reduces the number of times the thread will have to go looking for work, as it works on the biggest available chunks of work first.

2.2. *ForkJoinPool* Instantiation

In Java 8, the most convenient way to get access to the instance of the *ForkJoinPool* is to use its static method *commonPool (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool--).* As its name suggests, this will provide a reference to the common pool, which is a default thread pool for every *ForkJoinTask*.

According to Oracle's documentation (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html), using the predefined common pool reduces resource consumption, since this discourages the creation of a separate thread pool per task.

```
1 ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

The same behavior can be achieved in Java 7 by creating a *ForkJoinPool* and assigning it to a *public static* field of a utility class:

```
1 public static ForkJoinPool forkJoinPool = new ForkJoinPool(2);
```

Now it can be easily accessed:

```
ForkJoinPool forkJoinPool = PoolUtil.forkJoinPool;
```

With *ForkJoinPool's* constructors, it is possible to create a custom thread pool with a specific level of parallelism, thread factory, and exception handler. In the example above, the pool has a parallelism level of 2. This means that pool will use 2 processor cores.

3. *ForkJoinTask<V>*

ForkJoinTask is the base type for tasks executed inside *ForkJoinPool.* In practice, one of its two subclasses should be extended: the *RecursiveAction* for *void* tasks and the *RecursiveTask<V>* for tasks that return a value. They both have an abstract method *compute()* in which the task's logic is defined.

3.1. *RecursiveAction – An Example*

In the example below, the unit of work to be processed is represented by a *String* called *workload*. For demonstration purposes, the task is a nonsensical one: it simply uppercases its input and logs it.

To demonstrate the forking behavior of the framework, **the example splits the task if *workload.length()* is larger than a specified threshold** using the *createSubtask()* method.

The String is recursively divided into substrings, creating *CustomRecursiveTask* instances which are based on these substrings.

As a result, the method returns a *List<CustomRecursiveAction>.*

The list is submitted to the *ForkJoinPool* using the *invokeAll()* method:

```
public class CustomRecursiveAction extends RecursiveAction {

    private String workload = "";
    private static final int THRESHOLD = 4;

    private static Logger logger =
      Logger.getAnonymousLogger();

    public CustomRecursiveAction(String workload) {
        this.workload = workload;
    }

    @Override
    protected void compute() {
        if (workload.length() > THRESHOLD) {
            ForkJoinTask.invokeAll(createSubtasks());
        } else {
           processing(workload);
        }
    }

    private List<CustomRecursiveAction> createSubtasks() {
        List<CustomRecursiveAction> subtasks = new ArrayList<>();

        String partOne = workload.substring(0, workload.length() / 2);
        String partTwo = workload.substring(workload.length() / 2, workload.length());

        subtasks.add(new CustomRecursiveAction(partOne));
        subtasks.add(new CustomRecursiveAction(partTwo));

        return subtasks;
    }

    private void processing(String work) {
        String result = work.toUpperCase();
        logger.info("This result - (" + result + ") - was processed by "
          + Thread.currentThread().getName());
    }
}
```

This pattern can be used to develop your own *RecursiveAction* classes. To do this, create an object which represents the total amount of work, chose a suitable threshold, define a method to divide the work, and define a method to do the work.

3.2. *RecursiveTask<V>*

For tasks that return a value, the logic here is similar, except that resulting for each subtask is united in a single result:

```
public class CustomRecursiveTask extends RecursiveTask<Integer> {
    private int[] arr;

    private static final int THRESHOLD = 20;

    public CustomRecursiveTask(int[] arr) {
        this.arr = arr;
    }

    @Override
    protected Integer compute() {
        if (arr.length > THRESHOLD) {
            return ForkJoinTask.invokeAll(createSubtasks())
              .stream()
              .mapToInt(ForkJoinTask::join)
              .sum();
        } else {
            return processing(arr);
        }
    }

    private Collection<CustomRecursiveTask> createSubtasks() {
        List<CustomRecursiveTask> dividedTasks = new ArrayList<>();
        dividedTasks.add(new CustomRecursiveTask(
          Arrays.copyOfRange(arr, 0, arr.length / 2)));
        dividedTasks.add(new CustomRecursiveTask(
          Arrays.copyOfRange(arr, arr.length / 2, arr.length)));
        return dividedTasks;
    }

    private Integer processing(int[] arr) {
        return Arrays.stream(arr)
          .filter(a -> a > 10 && a < 27)
          .map(a -> a * 10)
          .sum();
    }
}
```

In this example, the work is represented by an array stored in the *arr* field of the *CustomRecursiveTask* class. The *createSubtask()* method recursively divides the task into smaller pieces of work until each piece is smaller than the threshold. Then, the *invokeAll()* method submits subtasks to the common pull and returns a list of *Future (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)*.

To trigger execution, the *join()* method called for each subtask.

In this example, this is accomplished using Java 8's *Stream API (https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html);* the *sum()* method is used as a representation of combining sub results into the final result.

4. Submitting Tasks to the *ForkJoinPool*

To submit tasks to the thread pool, few approaches can be used.

The ***su*** e cases are the same):

```
1  f                                    Task);
2  i                                    n();
```

The ***in*** its for the result, and doesn't need any manual joining:

```
1  i                                    tomRecursiveTask);
```

The ***in*** ient way to submit a sequence of *ForkJoinTasks* to the *ForkJo* two tasks, var args, or a collection), forks them returns a collect which they were produced.

Alterna **nd *join()*** methods. The *fork()* method submits a task to a pool, b e *join()* method is be used for this purpose. In the case of *Rec* g but *null*; for *RecursiveTask<V>*, it returns the result of the task's execution:


```
1  customRecursiveTaskFirst.fork();
2  res                                  oin();
```

In our *RecursiveTask<V>* example we used the *invokeAll()* method to submit a sequence of subtasks to the pool. The same job can be done with *fork()* and *join()*, though this has consequences for the ordering of the results.

To avoid confusion, it is generally a good idea to use *invokeAll()* method to submit more than one task to the *ForkJoinPool.*

5. Conclusions

Using the fork/join framework can speed up processing of large tasks, but to achieve this outcome, some guidelines should be followed:

- **Use as few thread pools as possible** – in most cases, the best decision is to use one thread pool per application or system
- **Use the default common thread pool,** if no specific tuning is needed
- **Use a reasonable threshold** for splitting *ForkJoingTask* into subtasks
- **Avoid any blocking in your *ForkJoingTasks***

The examples used in this article are available in the linked GitHub repository (https://github.com/eugenp/tutorials/tree/master/core-java).