Fork/Join Framework

The fork-join framework allows to break a certain task on several workers and then wait for the result to combine them. It leverages multi-processor machine's capacity to great extent. Following are the core concepts and objects used in fork-join framework.

The main weakness of the traditional executor service implementations when dealing with tasks, which in turn depend on other subtasks, is that a thread is not able to put a task back to the queue or to the side and then serves/executes an new task. The Fork/Join Framework addresses this limitation by introducing another layer between the tasks and the threads executing them, which allows the threads to put blocked tasks on the side and deal with them when all their dependencies are executed. In other words, if Task 1 depends on Task 6, which task (Task 6) was created by Task 1, then Task 1 is placed on the side and is only executed once Task 6 is executed. This frees the thread from Task 1, and allows it to execute other tasks, something which is not possible with the traditional executor service implementations.

This is achieved by the use of fork and join operations provided by the framework (hence the name Fork/Join). Task 1 forks Task 6 and then joins it to wait for the result. The fork operation puts Task 6 on the queue while the join operation allows Thread 1 to put Task 1 on the side until Task 6 completes. This is how the fork/join works, fork pushes new things to the queue while the join causes the current task to be sided until it can proceed, thus blocking no threads.

Fork -

Fork is a process in which a task splits itself into smaller and independent sub-tasks which can be executed concurrently.

if we split problem A into A1 and A2 then A1.fork() will be handled by one thread and A2.fork() will be handled by another one.

Join -

Join is a process in which a task join all the results of sub-tasks once the subtasks have finished executing, otherwise it keeps waiting.

ForkJoinPool -

it is a special thread pool designed to work with fork-and-join task splitting.

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

Here a new ForkJoinPool with a parallelism level of 4 CPUs.

RecursiveAction-

RecursiveAction represents a task which does not return any value.

```
class Writer extends RecursiveAction {
  @Override
  protected void compute() { }
```

}

RecursiveTask-

RecursiveTask represents a task which returns a value.

Syntax

```
class Sum extends RecursiveTask<Long> {
  @Override
  protected Long compute() { return null; }
}
```