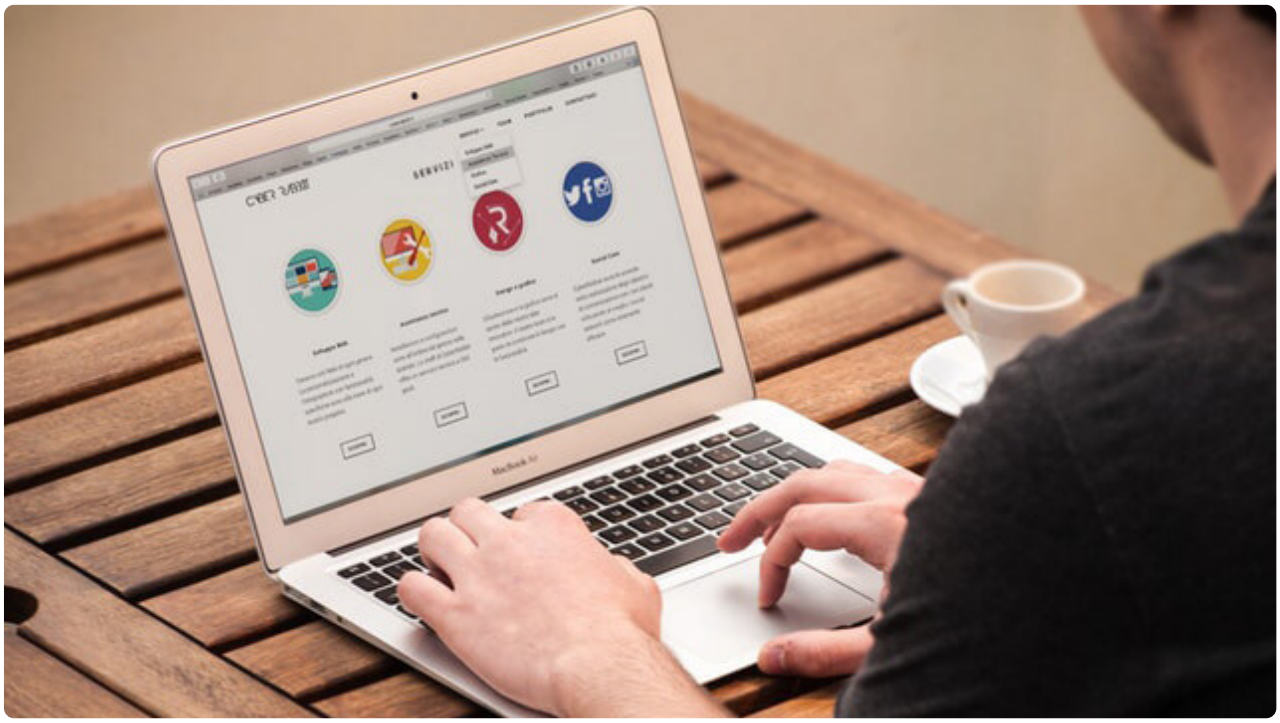
35拆分你的任务—学习使用Fork/Join框架

更新时间：2019-12-25 09:40:02

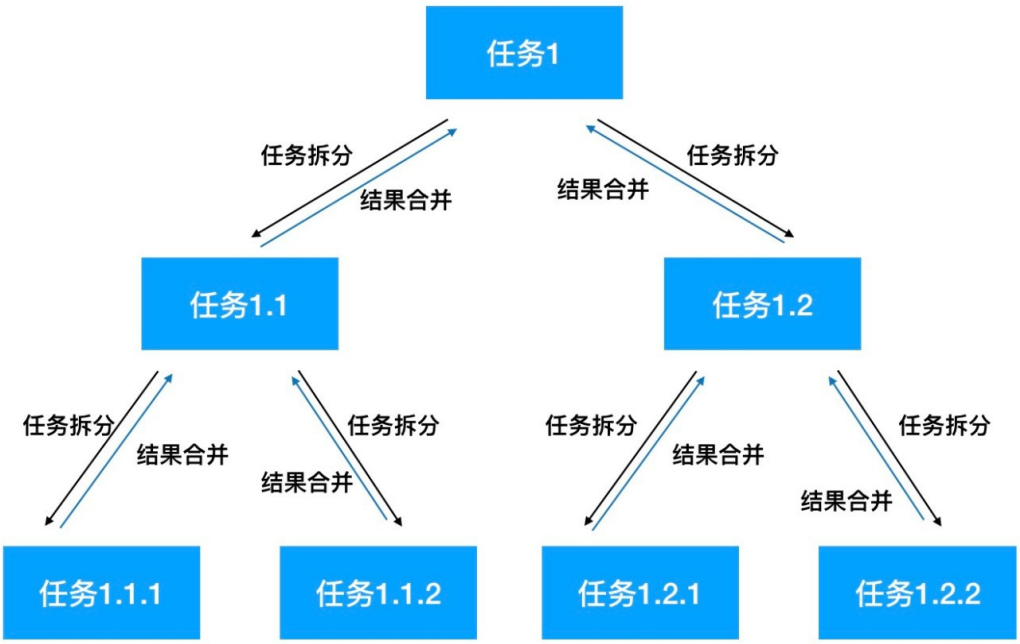


> 读书给人以快乐、给人以光彩、给人以才干。
>
> ——培根

本节我们学习 Excutor 的另外一种实现 ForkJoinPool。顾名思义，ForkJoinPool 的核心功能有两个。第一个是 Fork，拆解你的任务。第二个是 Join，合并任务的执行结果。这个场景很常见，比如我们要处理一批数据，由于数据间没有依赖性，那么我们可以把这一批数据拆解为更小的批次，多线程并行处理。最后再合并处理的结果。Fork/Join 的核心思想就是分而治之。

1、ForkJoinPool 介绍

ForkJoinPool 自 Java 7 引入。它和 ThreadPoolExecutor 都继承自 AbstractExecutorService，实现了 ExecutorService 和 Executor 接口。ForkJoinPool 用来把大任务切分为小任务，如果切分完小任务还不够小（由你设置的阈值决定），那么就继续向下切分。经过切分后，最后的任务是金字塔形状，计算完成后向上汇总。如下图：



ForkJoinPool 处理任务的核心思想可以用如下伪代码表示：

```
Result solve(Problem problem) {
 if (problem is small)
  directly solve problem
 else {
  split problem into independent parts
  fork new subtasks to solve each part
  join all subtasks
  compose result from subresults
 }
}
```

如果一个任务足够小，那么执行任务逻辑。如果不够小，拆分为两个独立的子任务。子任务执行后， 取得两个子任务的执行结果进行合并。

ForkJoinPool 通过 submit 执行 ForkJoinTask 类型的任务。ForkJoinTask 是抽象类，有着不同的子类实现。比较常用的是如下两种：

1、RecursiveAction，没有返回值；

2、RecurisiveTask，有返回值。

此外 submit 方法还可以执行 Callable 和 Runnable 的接口实现。

ForkJoinTask 就是我们为代码中的 problem。我来举个例子看具体如何使用。假如让你计算 1-10000 的和，

我们可以把任务拆解为 100 个，每个任务计算 100 个数字之和。代码如下.

Task 代码；

```
public class Task extends RecursiveTask<Integer> {

    private static final int THRESHOLD = 100;
    private int from;
    private int to;

    public Task(int from, int to) {
        super();
        this.from = from;
        this.to = to;
    }

    @Override
    protected Integer compute() {
        if (THRESHOLD > (to - from)) {
            return IntStream.range(from, to + 1)
                    .reduce((a, b) -> a + b)
                    .getAsInt();
        } else {
            int forkNumber = (from + to) / 2;
            Task left = new Task(from, forkNumber);
            Task right = new Task(forkNumber + 1, to);

            left.fork();
            right.fork();

            return left.join() + right.join();
        }
    }

}
```

Task 继承自 RecursiveTask。递归任务的大小力度为 100。重写的 compute 方法和文章开头的伪代码 solve 是一样的思路。先判断任务的大小是否在 THRESHOLD 之内。如果已经拆解到 THRESHOLD 内，那么进行计算。如果任务拆分还没达到 THRESHOLD，那么继续拆解任务。fork 操作会把当前任务放入线程池中来执行。最后再通过 join 取得执行结果做合并。

Client 代码：

```
public class Client {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
        ForkJoinTask<Integer> result = forkJoinPool.submit(new Task(1, 10000));

        System.out.println("计算结果为"+result.get());
        forkJoinPool.shutdown();
    }
}
```

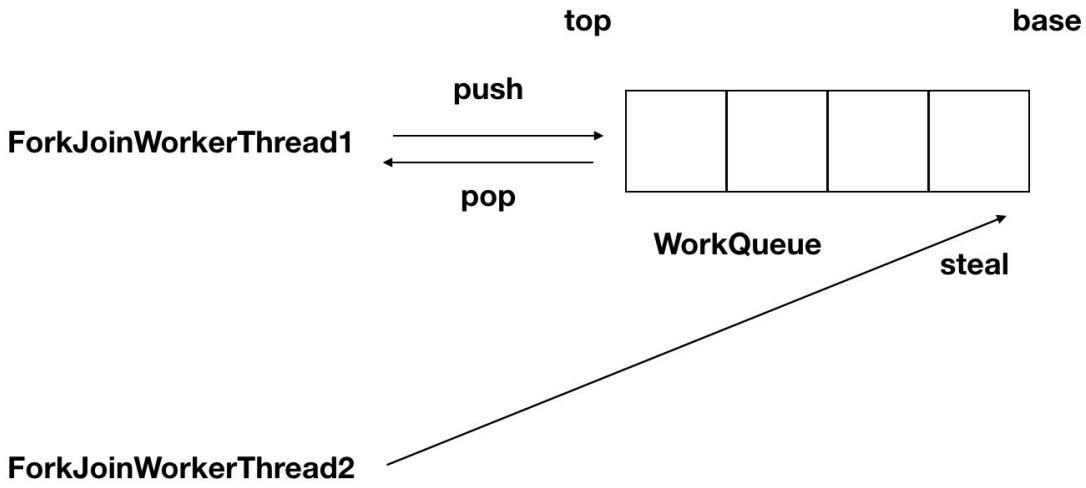
我们首先通过静态方法 commonPool 声明一个 ForkJoinPool。commonPool 创建的 ForkJoinPool 满足绝大多数的应用场景。然后通过 submit 方法提交我们的 Task，计算 1-10000 的和。提交 Task 后，Task 中的 compute 方法最终会被调用，通过对任务的拆解，以及对任务计算结果的合并，汇总到此处的 Task 中。通过 Task 的 get 方法获取计算结果。最后关闭线程池。

执行结果如下：

```
计算结果为50005000
```

2、ForkJoinPool 原理介绍

ForkJoinPool 中的每个线程都维护自己的工作队列。这是一个双端队列，既可以先进先出，也可以先进后出。简单来说就是队列两端都可以做出队操作。当每个线程产生新的任务时（比如说调用了 fork 操作），会被加入到队尾。线程工作的时候会从自己维护的工作队列的 top 做出队操作（LIFO），取得任务来执行。线程还会去其它线程任务队列窃取任务，此时是从其它队列的 base 取得任务（FIFO）。如下图所示：



下面简单介绍几个常用方法：

1、fork 方法中会判断如果当前线程不是 ForkJoinWorkerThread，则把任务加入 submission queue。否则加入自己的工作队列中。submission queue 没有关联的线程，是所有线程都可以执行的任务队列。fork 代码如下：

```
public final ForkJoinTask<V> fork() {
    Thread t;
  //判断本线程是否为ForkJoinWorkerThread，是的话，加入到自己的workQueue中，否则调用externalPush
    if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
        ((ForkJoinWorkerThread)t).workQueue.push(this);
    else
        ForkJoinPool.common.externalPush(this);
    return this;
}
```

2、join 方法中，自己任务没有执行完，则取的自己任务队列中的任务执行。如果发现自己的任务已经没有了，则会去窃取其它线程的任务来执行。Join 代码如下：

```
public final V join() {
    int s;
  //取得doJoin后的状态，位运算后判断是否正常，不正常的话抛出异常。正常的话返回计算结果
    if ((s = doJoin() & DONE_MASK) != NORMAL)
        reportException(s);
    return getRawResult();
}
```

主要逻辑在 doJoin 中，代码如下：

```
private int doJoin() {
    int s; Thread t; ForkJoinWorkerThread wt; ForkJoinPool.WorkQueue w;
    return (s = status) < 0 ? s :
        ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) ?
        (w = (wt = (ForkJoinWorkerThread)t).workQueue).
        tryUnpush(this) && (s = doExec()) < 0 ? s :
        wt.pool.awaitJoin(w, this, 0L) :
        externalAwaitDone();
}
```

如果当前线程不是 ForkJoinWorkerThread，则调用 externalAwaitDone。如果是 ForkJoinWorkerThread 那么先通过 tryUnpush 从自己的 workQueue 的 top 位置取得当前 task，然后调用 doExec 执行。这两步成功的话返回执行结果 s，否则调用 awaitJoin。这个方法中判断本任务是否执行完成，完成直接返回，否则会尝试窃执行取别的线程的任务。

3、submit 方法中，会把任务 push 到 submission queue。

ForkJoinPool 通过任务窃取，使得任务的执行更为高效。

3、总结

ForkJoinPool 为我们拆分大任务再汇总小任务计算结果提供了很好的支持。它很适合执行计算密集型的任务。但是如果你的任务拆分逻辑比计算逻辑还要复杂，ForkJoinPool 并不能为你带来性能的提升，反而会起到负面作用。因此需要结合自己的场景来选择使用。

}

34 谁都不能偷懒-通过 CompletableFuture 组装你的异步计算单元

36 为多线程们安排一位经理—Master/Slave模式详解