# 神奇好望角 The Magical Cape of Good Hope

blogjava.net/shinzey/archive/2012/02/09/368312.html

## 非主流並發工具之 ForkJoinPool

Posted on 2012-02-09 10:40 蜀山兆矛龘 閱讀(2155) 評論(2) 編輯 收藏 所屬分類: Java SE ·

ForkJoinPool 是 Java SE 7 新功能「分叉/結合框架」的核心類，現在可能乏人問津，但我覺得它遲早會成為主流。分叉/結合框架是一個比較特殊的線程池框架，專用於需要將一個任務不斷分解成子任務（分叉），再不斷進行彙總得到最終結果（結合）的計算過程。比起傳統的線程池類 ThreadPoolExecutor，ForkJoinPool 實現了工作竊取算法，使得空閒線程能夠主動分擔從別的線程分解出來的子任務，從而讓所有的線程都儘可能處於飽滿的工作狀態，提高執行效率。

ForkJoinPool 提供了三類方法來調度子任務：

execute 系列
　　異步執行指定的任務。
invoke 和 invokeAll
　　執行指定的任務，等待完成，返回結果。
submit 系列
　　異步執行指定的任務並立即返回一個 Future 對象。

子任務由 ForkJoinTask 的實例來代表。它是一個抽象類，JDK 為我們提供了兩個實現：RecursiveTask 和 RecursiveAction，分別用於需要和不需要返回計算結果的子任務。ForkJoinTask 提供了三個靜態的 invokeAll 方法來調度子任務，注意只能在 ForkJoinPool 執行計算的過程中調用它們。

ForkJoinPool 和 ForkJoinTask 還提供了很多讓人眼花繚亂的公共方法，其實它們大多數都是其內部實現去調用的，對於應用開發人員來說意義不大。

下面以統計 D 盤文件個數為例。這實際上是對一個文件樹的遍歷，我們需要遞歸地統計每個目錄下的文件數量，最後彙總，非常適合用分叉/結合框架來處理：

```
// 處理單個目錄的任務
public class CountingTask extends RecursiveTask<Integer> {
    private Path dir;

    public CountingTask(Path dir) {
        this.dir = dir;
    }

    @Override
    protected Integer compute() {
        int count = 0;
        List<CountingTask> subTasks = new ArrayList<>();

        // 讀取目錄 dir 的子路徑。
        try (DirectoryStream<Path> ds = Files.newDirectoryStream(dir)) {
            for (Path subPath : ds) {
                if (Files.isDirectory(subPath, LinkOption.NOFOLLOW_LINKS)) {
                    // 對每個子目錄都新建一個子任務。
```

```
                    subTasks.add(new CountingTask(subPath));
                } else {
                    // 遇到文件，則計數器增加 1。
                    count++;
                }
            }

            if (!subTasks.isEmpty()) {
                // 在當前的 ForkJoinPool 上調度所有的子任務。
                for (CountingTask subTask : invokeAll(subTasks)) {
                    count += subTask.join();
                }
            }
        } catch (IOException ex) {
            return 0;
        }
        return count;
    }
}

// 用一個 ForkJoinPool 實例調度「總任務」，然後敬請期待結果……
Integer count = new ForkJoinPool().invoke(new CountingTask(Paths.get("D:/")));
```

在我的筆記本上，經多次運行這段代碼，耗費的時間穩定在 600 豪秒左右。普通線程池（`Executors.newCachedThreadPool()`）耗時 1100 毫秒左右，足見工作竊取的優勢。

結束本文前，我們來圍觀一個最神奇的結果：單線程算法（使用 `Files.walkFileTree(...)`）比這兩個都快，平均耗時 550 毫秒！這警告我們並非引入多線程就能優化性能，並須要先經過多次測試才能下結論。