分解和合併：Java 也擅長輕鬆的並行編程！

oracle.com/technetwork/cn/articles/java/fork-join-422606-zhs.html

作者：Julien Ponge

Java SE 7 提供的新分解/合併任務如何使編寫並行程序變得更輕鬆？

2011 年 7 月發佈

下載：

Java SE 7

示例代碼 (Zip)

多核處理器現在已廣泛應用於服務器、台式機和便攜機硬件。它們還擴展到到更小的設備，如智能電話和平板電腦。由於進程的線程可以在多個內核上並行執行，因此多核處理器為並發編程打開了一扇扇新的大門。為實現應用程序的最大性能，一項重要的技術就是將密集型任務拆分成可以並行執行的若干小塊，以便最大程度利用計算能力。

傳統上，處理並發（並行）編程一直很困難，因為您不得不處理線程同步和共享數據的問題。Groovy (GPar)、Scala 和 Clojure 社區的努力已經證明，人們對 Java 平台上並發編程的語言級支持的興趣十分強烈。這些社區都嘗試提供全面的編程模型和高效的實現，以屏蔽與多線程和分佈式應用程序相關的難點。但不應認為 Java 語言本身在這方面遜色。Java Platform, Standard Edition (Java SE) 5 及後來的 Java SE 6 引入了一組程序包，可以提供強大的並發構建塊。Java SE 7 通過添加並行支持進一步增強了這些構建塊。

下文首先簡單回顧了 Java 並發編程，從早期版本以來已經存在的低級機制開始。然後在介紹 Java SE 7 中由分解/合併框架提供的新增基本功能分解/合併任務之前，先介紹 java.util.concurrent 程序包添加的豐富基元。文中給出了這些新 API 的示例用法。最後，在結束之前對方法進行了討論。

下面，我們假定讀者擁有 Java SE 5 或 Java SE 6 編程背景。在此過程中，我們還將介紹 Java SE 7 一些實用的語言發展。

Java 並發編程

傳統線程

過去，Java 並發編程包括通過 java.lang.Thread 類和 java.lang.Runnable 接口編寫線程，然後確保其代碼以正確、一致的方式對共享可變對象進行操作並避免錯誤的讀/寫操作，同時不會產生由於鎖爭用條件所導致的死鎖。以下是基本線程操作的示例：

```
Thread thread = new Thread() {
      @Override public void run() {
         System.out.println(">>> I am running in a separate thread!");
      }
 };
 thread.start();
 thread.join();
```

本示例中的代碼所做的只是創建一個線程，該線程將一個字符串打印到標準輸出流。主線程通過調用 join() 等待所創建的（子）線程完成。

這樣直接操作線程對於簡單示例來說是不錯，但對於並發編程，這種代碼很快就容易產生錯誤，尤其是當多個線程需要合作執行一個大型任務時。在這樣的情況下，需要協調其控制流。

例如，某個線程執行的完成可能依賴於其他線程執行完成。通常人們熟知的示例是生產者/使用者的例子，如果使用者的隊列已滿則生產者應等待使用者，當隊列為空時使用者應等待生產者。這一要求可通過共享狀態和條件隊列得到滿足，但您仍需要通過對共享狀態對象使用 java.lang.Object.notify() 和 java.lang.Object.wait() 來使用同步，這很容易出錯。

最後，一個常見的問題是對大段代碼甚至是整個方法使用同步和提供互斥。儘管此方法可產生線程安全的代碼，但由於排除實際上過長所引起的有限並行度，該方法通常導致性能變差。

正如計算中經常發生的那樣，操作低級基元以實現複雜操作會打開錯誤之門，因此開發人員應想辦法將複雜性封裝在高效的高級庫中。Java SE 5 正好為我們提供了這種能力。

java.util.concurrent 程序包的豐富基元

Java SE 5 引入了一個名為 java.util.concurrent 的程序包系列，Java SE 6 對其進行了進一步的增強。該程序包系列提供了以下並發編程基元、集合和特性：

- 執行器是對傳統線程的增強，因為它們是從線程池管理抽象而來的。它們執行與傳遞到線程的任務類似的任務（實際上，可封裝實現 java.lang.Runnable 的實例）。有些實現提供了線程池和調度策略。而且，可以通過同步和異步方式獲取執行結果。
- 線程安全隊列允許在並發任務之間傳遞數據。底層數據結構和並發行為有著豐富的實現，底層數據結構的實現包括數組列表、鏈接列表或雙端隊列等，並發行為的實現包括阻塞、支持優先級或延遲等。
- 細粒度的超時延遲規範，因為 java.util.concurrent 程序包中的大部分類均支持超時延遲。例如，如果任務在規定時間範圍內無法完成，執行器將中斷任務執行。
- 豐富的同步模式，不僅僅是 Java 中低級同步塊所提供的互斥。這些模式包括信號或同步障礙等常用語法。
- 高效、並發的數據集合（映射、列表和集），通過使用寫時複製和細粒度鎖通常可在多線程上下文中產生出色的性能。
- 原子變量，可以使開發人員免於親自執行同步訪問。這些變量封裝了常用的基元類型，如整型或布爾值，以及對其他對象的引用。
- 超出固有鎖所提供的鎖定/通知功能範圍的多種鎖，例如，支持重新進入、讀/寫鎖定、超時或基於輪詢的鎖定嘗試。

例如，考慮以下程序：

注意：由於 Java SE 7 引入的新的整數文本，可以在任意位置插入下劃線以提高可讀性（例如，1_000_000）。

```
import java.util.*;
import java.util.concurrent.*;
import static java.util.Arrays.asList;

public class Sums {

    static class Sum implements Callable<Long> {
        private final long from;
        private final long to;
            Sum(long from, long to) {
```

```
            this.from = from;
            this.to = to;
        }

        @Override
        public Long call() {
            long acc = 0;
            for (long i = from; i <= to; i++) {
                acc = acc + i;
            }
            return acc;
        }
    }

    public static void main(String[] args) throws Exception {

        ExecutorService executor = Executors.newFixedThreadPool(2);
        List <Future<Long>> results = executor.invokeAll(asList(
            new Sum(0, 10), new Sum(100, 1_000), new Sum(10_000, 1_000_000)
        ));
        executor.shutdown();

        for (Future<Long> result : results) {
            System.out.println(result.get());
        }
    }
}
```

該示例程序利用執行器來計算多個長整型的和。內部 Sum 類實現了執行器用於計算結果的 Callable 接口，並發工作在 call() 方法內執行。java.util.concurrent.Executors 類提供了多種實用方法，如提供預配置執行器或將傳統 java.lang.Runnable 對象封裝到 Callable 實例中。與 Runnable 相比，使用 Callable 的優勢在於 Callable 能夠顯式返回一個值。

本示例使用一個執行器將工作分派給兩個線程。ExecutorService.invokeAll() 方法接受 Callable 實例的集合，並在返回之前等待所有這些實例完成。它會返回 Future 對象的列表，這些對象全都表示計算的「未來」結果。如果我們以異步方式工作，就可以測試每個 Future 對象來檢查其對應的 Callable 是否已完成工作，並檢查其是否引發了異常，甚至可以取消其工作。相反，當使用普通傳統線程時，必須通過共享的可變布爾值對取消邏輯進行編碼，並由於定期檢查此布爾值而減緩代碼的執行。因為 invokeAll() 容易產生阻塞，我們可以直接對 Future 實例進行遍歷並讀取其計算和。

還需注意，必須關閉執行器服務。如果未關閉，則在主方法退出時 Java 虛擬機將不會退出，因為環境中還有活動線程。

分解/合併任務

概述

與傳統線程相比，執行器是一大進步，因為可以簡化並發任務的管理。有些類型的算法要求任務創建子任務並與其他任務互相通信以完成任務。這些是「分而治之」的算法，也稱為「映射歸約」，類似函數語言中的齊名函數。其思路

是將算法要處理的數據空間拆分成較小的獨立塊。這是「映射」階段。一旦塊集處理完畢之後，就可以將部分結果收集起來形成最終結果。這是「歸約」階段。

一個簡單的示例是您希望計算一個大型整數數組的總和（參見圖 1）。假定加法是可交換的，可以將數組劃分為較小的部分，並發線程對這些部分計算部分和。然後將部分和相加，計算總和。因為對於此算法，線程可以在數組的不同區域上獨立運行，所以與對數組中每個整數循環執行的單線程算法相比，此算法在多核架構上可以看到明顯的性能提升。

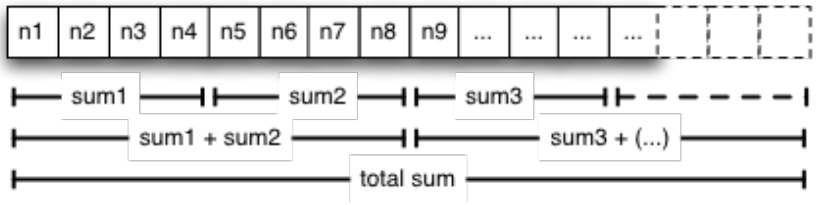


圖 1：整數數組的部分和

使用執行器解決以上問題很簡單：將數組分為 n 個可用物理處理單元，創建 Callable 實例以計算每個部分和，將部分和提交給管理 n 個線程的線程池的執行器，然後收集結果以計算最終和。

但對於其他類型的算法和數據結構，執行計畫通常不會如此簡單。尤其是，標識「足夠小」可通過高效方式獨立處理的數據塊的「映射」階段預先不知道數據空間拓撲結構。對基於圖形和基於樹的數據結構來說尤為如此。在這些情況下，算法應創建「各部分」的層次結構，在返回部分結果之前等待子任務完成。儘管類似圖 1 中的數組並非最優，但可以使用多級並發部分和計算（例如，在雙核處理器上將數組分為 4 個子任務）。

用於實現分而治之算法的執行器的問題與創建子任務無關，因為 Callable 可自由向其執行器提交新的子任務，然後以同步或異步方式等待其結果。問題出在並行上：當 Callable 等待另一個 Callable 的結果時，它被置於等待狀態，因此浪費了處理排隊等待執行的另一個 Callable 的機會。

通過 Doug Lea 的努力，在 Java SE 7 中添加到 java.util.concurrent 程序包的分解/合併框架填補了這一空白。Java SE 5 和 Java SE 6 版本的 java.util.concurrent 幫助處理並發，Java SE 7 中另外增加了一些功能幫助處理並行。

用於支持並行的新增功能

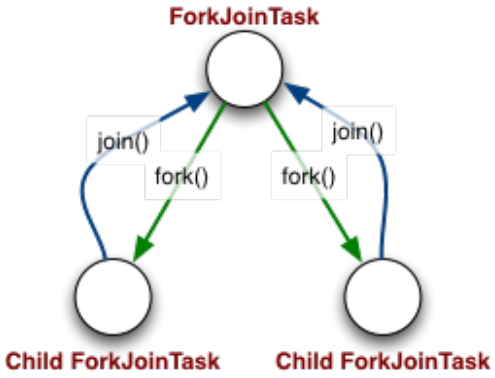
核心新增功能是專用於運行實現 ForkJoinTask 實例的新的 ForkJoinPool 執行器。ForkJoinTask 對象支持創建子任務並等待子任務完成。通過這些明確的語義，執行器能夠通過在任務等待另一任務完成並且有待處理任務要運行時「竊取」作業，從而在其內部線程池中分派任務。

ForkJoinTask 對象有兩種特定方法：

- fork() 方法允許計畫 ForkJoinTask 異步執行。這允許從現有 ForkJoinTask 啟動新的 ForkJoinTask。
- 而 join() 方法允許 ForkJoinTask 等待另一個 ForkJoinTask 完成。

任務之間的合作通過 fork() 和 join() 來實現，如圖 2 所示。請注意，fork() 和 join() 方法名不應與其 POSIX 對應項（進程可通過它複製自身）混淆。其中，fork() 僅在 ForkJoinPool 中調度一個新任務，但不創建子 Java 虛擬機。

圖 2：Fork 和 Join 任務之間的合作



有兩種類型的 ForkJoinTask 實現：

- RecursiveAction 的實例表示不產生返回值的執行。
- 相反，RecursiveTask 的實例會產生返回值。

通常，優先選擇 RecursiveTask，因為大多數的分而治之算法返回數據集的計算值。對於任務的執行，提供了不同的同步和異步選項，從而有可能實現細緻的模式。

示例：計算某個單詞在文檔中出現的次數

為了說明新的分解/合併框架的用法，我們舉一個簡單示例：計算某個單詞在一組文檔中出現的次數。首先，分解/合併任務應作為「純」內存中算法運行，其中不涉及 I/O 操作。同時，應儘可能避免任務之間通過共享狀態的通信，因為這意味著可能必須執行鎖定。理想情況下，僅當一個任務分出另一個任務或一個任務併入另一個任務時，任務之間才進行通信。

我們的應用程序運行在文件目錄結構上，將每個文件的內容加載到內存中。因此，需要以下類來表示該模型。文檔表示為一系列行：

```
class Document {
    private final List<String> lines;

    Document(List<String> lines) {
        this.lines = lines;
    }

    List<String> getLines() {
        return this.lines;
    }

    static Document fromFile(File file) throws IOException {
        List<String> lines = new LinkedList<>();
        try(BufferedReader reader = new BufferedReader(new FileReader(file))) {
            String line = reader.readLine();
            while (line != null) {
                lines.add(line);
                line = reader.readLine();
            }
        }
        return new Document(lines);
    }
}
```

注意：如果您是初次接觸 Java SE7，fromFile() 方法有兩點會使您感到驚訝：

- LinkedList 使用尖括號語法 (<>) 告知編譯器推斷通用類型參數。由於行是 List，LinkedList<> 擴展為 LinkedList。使用尖括號運算符，對於那些能在編譯時輕鬆推斷的類型就不必再重複，從而使得通用類型的處理更輕鬆。
- try 塊使用新的自動資源管理語言特性。在 try 塊的開頭可以使用實現 java.lang.AutoCloseable 的任何類。無論是否引發異常，當執行離開 try 塊時，在此聲明的任何資源都將正常關閉。在 Java SE 7 之前，正常關閉多個資源很快會變成一場嵌套 if/try/catch/finally 塊的夢魘，這種嵌套塊通常很難正確編寫。

於是文件夾成為一個簡單的基於樹的結構：

```
class Folder {
    private final List<Folder> subFolders;
    private final List<Document> documents;

    Folder(List<Folder> subFolders, List<Document> documents) {
        this.subFolders = subFolders;
        this.documents = documents;
    }

    List<Folder> getSubFolders() {
        return this.subFolders;
    }

    List<Document> getDocuments() {
        return this.documents;
    }

    static Folder fromDirectory(File dir) throws IOException {
        List<Document> documents = new LinkedList<>();
        List<Folder> subFolders = new LinkedList<>();
        for (File entry : dir.listFiles()) {
            if (entry.isDirectory()) {
                subFolders.add(Folder.fromDirectory(entry));
            } else {
                documents.add(Document.fromFile(entry));
            }
        }
        return new Folder(subFolders, documents);
    }
}
```

現在我們可以開始實現主類：

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

public class WordCounter {
```

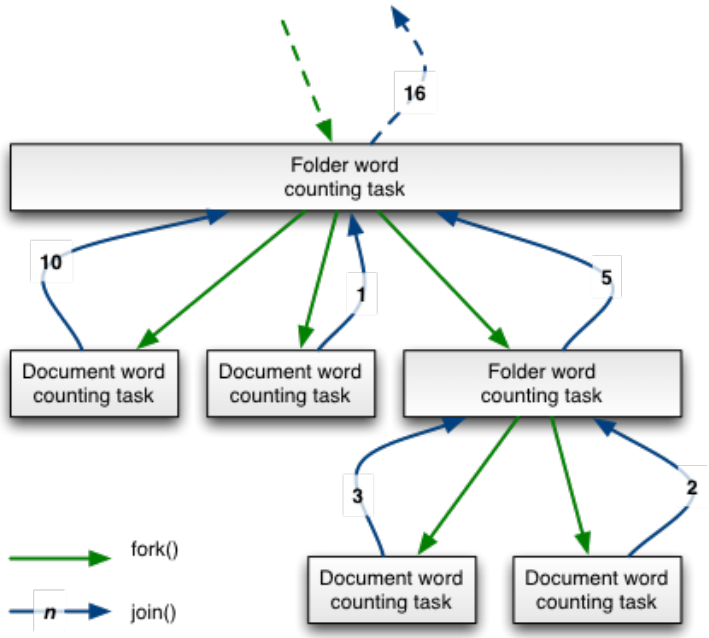
```
    String[] wordsIn(String line) {
        return line.trim().split("(\\s|\\p{Punct})+");
    }

    Long occurrencesCount(Document document, String searchedWord) {
        long count = 0;
        for (String line : document.getLines()) {
            for (String word : wordsIn(line)) {
                if (searchedWord.equals(word)) {
                    count = count + 1;
                }
            }
        }
        return count;
    }
}
```

occurrencesCount 方法利用 wordsIn 方法返回某個單詞在文檔中出現的次數，wordsIn 方法在一行中生成該單詞的數組。為此，該方法基於空格和標點字符對行進行拆分。

我們將實現兩種類型的分解/合併任務。直觀地說，一個文件夾中某個單詞出現的次數是該單詞在每個子文件夾和文檔中出現的次數的總和。因此，我們將有一個任務用於計算文檔中出現的次數，還有一個任務用於計算文件夾中出現的次數。後一類型分出子任務，然後將子任務合併以收集這些子任務的結果。

任務相關性易於掌握，因為它直接映射底層文檔或文件夾樹結構，如圖 3 中所示。分解/合併框架通過確保可以在文件夾任務等待 join() 操作時執行一個待處理文檔或文件夾的字數計算任務來使並行最大化。

圖 3：分解/合併字數計算任務



首先介紹 DocumentSearchTask，它計算某個單詞在文檔中出現的次數：

```
class DocumentSearchTask extends
RecursiveTask<Long> {
    private final Document document;
    private final String searchedWord;

    DocumentSearchTask(Document document,
String searchedWord) {
        super();
        this.document = document;
        this.searchedWord =
searchedWord;
    }

    @Override
    protected Long compute() {
        return occurrencesCount(document, searchedWord);
    }
}
```

因為我們的任務產生值，這些任務擴展了 RecursiveTask 並接受 Long 作為通用類型，因為出現的次數將由一個 long 型整數表示。compute() 方法是所有 RecursiveTask 的核心。此處，它只是委託上述 occurrencesCount() 方法。現在我們可以處理 FolderSearchTask 的實現，該任務運行在樹結構的文件夾元素上：

```
class FolderSearchTask extends RecursiveTask<Long> {
    private final Folder folder;
    private final String searchedWord;

    FolderSearchTask(Folder folder, String searchedWord) {
        super();
        this.folder = folder;
        this.searchedWord = searchedWord;
    }

    @Override
    protected Long compute() {
        long count = 0L;
        List<RecursiveTask<Long>> forks = new LinkedList<>();
        for (Folder subFolder : folder.getSubFolders()) {
            FolderSearchTask task = new FolderSearchTask(subFolder, searchedWord);
            forks.add(task);
            task.fork();
       }
        for (Document document : folder.getDocuments()) {
            DocumentSearchTask task = new DocumentSearchTask(document,
searchedWord);
            forks.add(task);
            task.fork();
       }
        for (RecursiveTask<Long> task : forks) {
            count = count + task.join();
       }
    return count;
  }
}
```

在該任務中，compute() 方法的實現只是為已通過其構造函數傳遞的每個文件夾元素分解文檔和文件夾任務。然後將合併所有這些任務以計算其部分和並返回該部分和。

現在我們只差一種方法來啟動分解/合併框架上的字數計算操作，以及一個分解/合併池執行器：

```
private final ForkJoinPool forkJoinPool = new ForkJoinPool();

Long countOccurrencesInParallel(Folder folder, String searchedWord) {
    return forkJoinPool.invoke(new FolderSearchTask(folder, searchedWord));
}
```

初始 FolderSearchTask 將啟動所有這些操作。ForkJoinPool 的 invoke() 方法允許等待計算完成。在上面的例子中，

ForkJoinPool 是通過其空構造函數來使用的。並行度將與可用的硬件處理單元的數目相匹配（例如，在具有雙核處理器的計算機上並行度將為 2）。

現在我們可以編寫一個 main() 方法，該方法從命令行參數接受要在其上運行的文件夾以及要搜索的字：

```
public static void main(String[] args) throws IOException {
    WordCounter wordCounter = new WordCounter();
    Folder folder = Folder.fromDirectory(new File(args[0]));
    System.out.println(wordCounter.countOccurrencesOnSingleThread(folder, args[1]));
}
```

該示例的完整源代碼還包括此算法更傳統的基於遞歸的實現，該實現工作在單線程上：

```
Long countOccurrencesOnSingleThread(Folder folder, String searchedWord) {
    long count = 0;
    for (Folder subFolder : folder.getSubFolders()) {
        count = count + countOccurrencesOnSingleThread(subFolder, searchedWord);
    }
    for (Document document : folder.getDocuments()) {
       count = count + occurrencesCount(document, searchedWord);
    }
   return count;
}
```

討論

在 Oracle 的 Sun Fire T2000 服務器上進行了一次非正式測試，其中可以指定 Java 虛擬機可用的內核數。同時運行了上例的分解/合併和單線程形式以找出 import 在 JDK 源代碼文件中出現的次數。

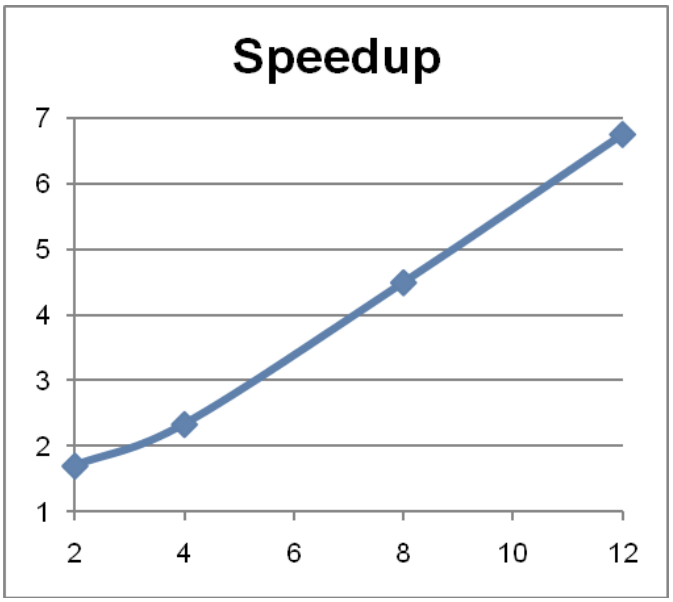
多次運行了這些變化形式以確保 Java 虛擬機熱點優化有足夠的時間完成部署。收集了 2 個、4 個、8 個和 12 個內核的最佳執行時間，然後計算了加速（即單線程上的時間/分解-合併上的時間之比）。結果反映在圖 4 和表 1 中。

正如您看到的，只需極少的努力即可在內核數上獲得近乎線性的加速，因為分解/合併框架會負責最大化並行度。

表 1：非正式測試執行時間和加速

內核數	單線程執行時間 (ms)	分解/合併執行時間 (ms)	加速
2	18798	11026	1.704879376
4	19473	8329	2.337975747
8	18911	4208	4.494058935
12	19410	2876	6.748956885

圖 4：加速（縱軸）與內核數（橫軸）有關



我們還可以對計算進行優化，分解任務使其在行級而不是在文檔級運行。這將使並發任務有可能在同一文檔的不同行上運行。但這有點牽強。實際上，分解/合併任務應執行「足夠」數量的計算以克服分解/合併線程池和任務管理的開銷。在行級工作將過於瑣碎，從而影響該方法的效率。

所包含的源代碼還提供基於對整數數組執行合併-排序算法的另一個分解/合併示例。這很有趣，因為它使用 RecursiveAction 來實現，該分解/合併任務對 join() 方法的調用不會產生值。相反，任務將共享可變狀態：要排序的數組。同樣，實驗顯示內核數目上存在近乎線性的加速。

總結

本文討論了 Java 並發編程，重點強調 Java SE 7 為簡化並行程序編寫而提供的新的分解/合併任務。本文顯示，可以使用和組合豐富的基元來編寫可利用多核處理器的高性能程序，而完全無需處理線程和共享狀態同步的低級操作。本文在某單詞出現次數計算示例中闡釋了這些新 API 的使用，既引人注目又易於掌握。在非正式測試中，在內核數目上取得了近乎線性的加速。這些結果顯示分解/合併框架非常有用；因為我們既不必更改代碼，也不必調整代碼或 Java 虛擬機，即可最大程度利用硬件內核。

您還可以將此技術應用於自己的問題和數據模型。只要您按無需 I/O 工作和鎖定的「分而治之」的方式重新編寫算法，即可看到顯著的加速。

致謝

作者要感謝 Brian Goetz 和 Mike Duigou 對本文的早期版本提供了非常有用的反饋。還要感謝 Scott Oaks 和 Alexis Moussine-Pouchkine 幫助在適當的硬件上運行測試。

另請參見

JavSE 下載：http://www.oracle.com/technetwork/cn/java/javase/downloads/index.html
示例代碼：http://www.oracle.com/technetwork/articles/java/forkjoinsources-430155.zip
Java SE 7 API：http://download.java.net/jdk7/docs/api/
JSR-166 興趣站點，作者 Doug Lea：http://gee.cs.oswego.edu/dl/concurrency-interest/
Coin 項目：http://openjdk.java.net/projects/coin/
《Java Concurrency in Practice》，作者：Brian Goetz、Tim Peierls、Joshua Bloch、Joseph Bowbeer、David Holmes 和 Doug Lea (Addision-Wesley Professional)：
http://www.informit.com/store/product.aspx?isbn=0321349601
合併-排序算法：http://en.wikipedia.org/wiki/Merge_sort
Groovy：http://groovy.codehaus.org/
GPar：http://gpars.codehaus.org/
Scala：http://scala-lang.org
Clojure：http://clojure.org/

Julien Ponge 是一位長期從事開源工作的技術高人。他創建了 IzPack 安裝程序框架，還參與了其他幾個項目，包括與 Sun Microsystems 合作的 GlassFish 應用服務器。他擁有 UNSW Sydney 和 UBP Clermont-Ferrand 的計算科學博士學位，目前是 INSA de Lyon 計算機科學系的副教授，並且是 INRIA Amazones 團隊的研究人員。他在行業和學術界的兩棲身份給了他極大的動力來進一步推進這兩個領域的合作。