**Sử dụng Fork/Join Framework với ForkJoinPool trong Java**

**Giới thiệu Fork/ Join**

[**Fork/ Join Framework**](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html) được thêm vào Java 7. Nó cung cấp các công cụ giúp tăng tốc xử lý song song bằng cách cố gắng sử dụng tất cả các lõi bộ xử lý có sẵn, được thực hiện thông qua cách tiếp cận phân chia (fork) và gộp (join) task. Mục đích là để sử dụng tất cả các khả năng xử lý để nâng cao hiệu suất cho các ứng dụng.

* Trong thực tế, bước đầu tiên framework Fork/ Join thực hiện là chia nhỏ task (fork/ split), [đệ quy](https://gpcoder.com/2327-giai-thuat-de-quy-trong-java/) chia nhỏ nhiệm vụ thành các nhiệm vụ phụ nhỏ hơn cho đến khi chúng đơn giản đủ để được thực hiện xử lý không đồng bộ.
* Sau đó, phần gộp kết quả (join) bắt đầu, trong đó các kết quả của tất cả các nhiệm vụ phụ được đệ quy một cách đệ quy vào một kết quả, hoặc trong trường hợp một nhiệm vụ trả về void, chương trình chỉ cần đợi cho đến khi mỗi nhiệm vụ phụ được thực hiện.

Có thể mô tả code như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | if (tác vụ nhỏ) {                    Trực tiếp giải quyết/ trả kết quả    } else {                    Chia nhỏ/ Tách tác vụ thành các tác vụ con                    Đệ quy giải quyết từng tác vụ con (fork)                    Tổng hợp kết quả (join)    } |

Để cung cấp thực hiện xử lý song song hiệu quả, các fork / join Framework sử dụng hồ chứa (pool) các Thread được gọi là ForkJoinPool.

**ForkJoinPool** tương tự như Java [ExecutorService](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/) nhưng với một sự khác biệt. ForkJoinPool phân chia các tác vụ cho các luồng thực thi trong Thread Pool. Framework Fork/ Join sử dụng thuật toán [work-stealing](https://en.wikipedia.org/wiki/Work_stealing). Các luồng sẽ thực thi công việc của mình trên một bộ xử lý riêng biệt (thread/ processor), khi làm hết việc của mình, nó lấy bớt (steal) các tác vụ từ các luồng khác đang bận rộn.

**Work stealing là gì?**

[Work stealing](https://en.wikipedia.org/wiki/Work_stealing) là cơ chế giúp scheduler (có thể là trên ngôn ngữ, hoặc OS) có thể thực hiện việc tạo thên M thread mới hoạt động mượt mà trên N core, với M có thể lớn hơn N rất nhiều.

Idea của work-stealing scheduler là mỗi một core sẽ có một queue những task phải làm. Mỗi task đó bao gồm một list các instructions phải thực hiện một cách tuần tự. Khi một processor làm hết việc của mình, nó sẽ nhìn ngó sang các processor xung quanh, xem có gì cần làm không và “steal” công việc từ đó.

Một mô hình khác với work stealing là work sharing, tức là mỗi task sẽ quyết fix là sẽ được thực hiện trên processor nào.

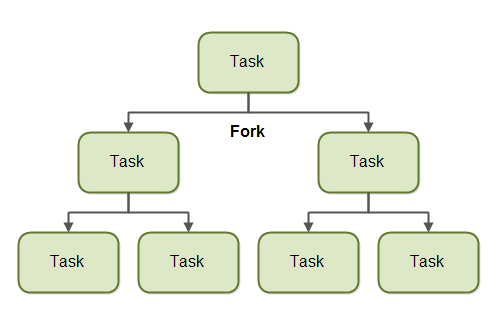
**Giải thích Fork and Join**

Trước khi vào phần chi tiết về ForkJoinPool, tôi sẽ giải thích đôi chút về nguyên tắc làm việc của Fork/ Join Framework.

Nguyên tắc Fork/ Join gồm 2 bước được thực hiện đệ quy. Hai bước này là: bước chia tách (fork/ split) và bước gộp (join/ merge).

**Nguyên tắc hoạt động của Fork**

Một nhiệm vụ (parent task) sử dụng nguyên tắc fork và join có thể chia tách (fork/ split) chính nó vào các nhiệm vụ con (sub task) nhỏ hơn để có thể được thực hiện đồng thời. Điều này được minh họa trong sơ đồ dưới đây:

[](https://gpcoder.com/wp-content/uploads/2018/03/java-fork-and-join-1.png)

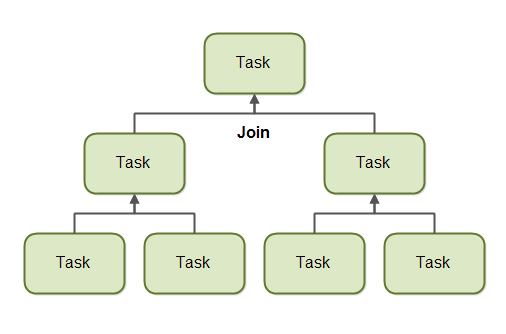
Bằng cách chia nhỏ thành các nhiệm vụ con, mỗi nhiệm vụ con có thể được thực hiện song song bởi các CPU khác nhau, hoặc các luồng khác nhau trên cùng một CPU.

Một nhiệm vụ chỉ phân chia thành các nhiệm vụ phụ nếu công việc mà nhiệm vụ được đưa ra là đủ lớn để điều này có ý nghĩa. Có một chi phí để chia tách một nhiệm vụ thành các nhiệm vụ phụ, vì vậy với số lượng nhỏ công việc trên không thể lớn hơn tốc độ đạt được bằng cách thực hiện các công việc phụ đồng thời.

**Nguyên tắc hoạt động của Join**

Khi một nhiệm vụ (parent task) đã tự tách mình thành các nhiệm vụ con (sub task), nhiệm vụ cha sẽ đợi cho đến khi các nhiệm vụ con hoàn thành.

Khi nhiệm vụ con đã hoàn thành, nhiệm vụ cha có thể kết hợp (join/ merge) tất cả các kết quả con vào một kết quả cuối cùng. Điều này được minh họa trong sơ đồ dưới đây:

[](https://gpcoder.com/wp-content/uploads/2018/03/java-fork-and-join-2.png)

Tất nhiên, không phải tất cả các loại nhiệm vụ có thể trả về một kết quả. Nếu các tác vụ không trả lại kết quả, thì một nhiệm vụ chỉ cần đợi cho các công việc phụ của nó hoàn thành, không có sự kết hợp kết quả nào xảy ra sau đó.

**ForkJoinPool**

Các ForkJoinPool là một [Thread Pool](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/) đặc biệt được thiết kế để làm việc tốt với chia tách công việc fork/ join. ForkJoinPool nằm trong gói java.util.concurrent, vì vậy tên lớp đầy đủ là **java.util.concurrent.ForkJoinPool**. Một số lớp tiêu biểu của Fork/ Join Framework:

* **ForkJoinTask<V>**: một abstract class định nghĩa task sẽ được thực thi trong một ForkJoinPool.
* **ForkJoinPool**: là một thread pool quản lý việc thực thi các ForkJoinTasks.
* **RecursiveAction**: là một lớp con của ForkJoinTask, nó thực thi tác vụ mà không trả lại bất kỳ kết quả nào (action).
* **RecursiveTask<V>**: là một lớp con của ForkJoinTask, nó thực thi tác vụ mà có trả lại kết quả (task).

Cú pháp tạo ForkJoinPool:

|  |  |
| --- | --- |
| 1 | ForkJoinPool forkJoinPool = new ForkJoinPool(numOfProcessor); |

Tạo một ForkJoinPool với tham số là số lượng luồng hoặc các CPU bạn muốn làm việc đồng thời trên các nhiệm vụ được truyền vào ForkJoinPool. Nếu bạn không xác định numOfProcessor, nó sẽ lấy số bộ vi xử lý có sẵn cho máy ảo Java để thực thi.

**Ví dụ sử dụng Fork/ Join Framework với ForkJoinPool**

Ví dụ viết chương trình tính tổng kích thước của folder. Tôi thực hiện 2 bước như sau:

* Tạo lớp tính kích thước một folder:
  + Sử dụng **RecursiveTask<T>**: để nhận kết quả tính toán.
  + Sử dụng **RecursiveAction**: lớp này không trả kết quả tính toán về, nên cần sử dụng một biến trung gian để lưu trữ kết quả.
* Viết chương trình sử dụng lớp ở trên để hiển thị kích thước của các folder với **ForkJoinPool**.

**Ví dụ sử dụng RecursiveTask**

Cú pháp khai báo một **RecursiveTask** như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class RecursiveTaskImpl extends RecursiveTask<T> {      @Override      protected T compute() {            }  } |

Ý tưởng giải quyết bài toán này là:

* Tạo lớp tính tổng kích thước của một folder kế thừa từ **abstract class RecursiveTask<T>**. Nó yêu cầu bạn phải **override** lại phương thức **compute()**. Phương thức này tương tự như **run()** của class **Thread** hay **call()** của interface **Callable**, khi Thread start() nó sẽ gọi phương thức này để xử lý.
* Phương thức **compute()** có kiểu trả về là một GenericType T. Kết quả xử lý của task sẽ được trả về từ phương thức này.
* Với yêu cầu này, đầu tiên chúng ta sẽ kiểm tra tham số truyền vào có phải là folder không, nếu không phải (tức là file) thì ngay lập tức return kích thước của file và kết thúc xử lý.
* Tiếp theo, nếu là folder, chúng ta sẽ đệ quy để tính toán kích thước của các folder con (bước này gọi là fork).
* Cuối cùng, cộng tất cả các kích thước trả về từ việc tính toán các folder con (bước này gọi là join).

Ta có chương trình xử lý như sau

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | package com.gpcoder.threadpool.forkjoin;    import java.io.File;  import java.util.ArrayList;  import java.util.List;  import java.util.Objects;  import java.util.concurrent.RecursiveTask;    public class SizeOfFileTask extends RecursiveTask<Long> {        private static final long serialVersionUID = -196522408291343951L;        private final File file;        public SizeOfFileTask(final String fileName) {          this(new File(fileName));      }        public SizeOfFileTask(final File file) {          this.file = Objects.requireNonNull(file);      }        @Override      protected Long compute() {          // System.out.printf("Computing size of: %s \n", file);            if (file.isFile()) {              return file.length();          }            final List<SizeOfFileTask> tasks = new ArrayList<>();          final File[] children = file.listFiles();          if (children != null) {              for (final File child : children) {                  final SizeOfFileTask task = new SizeOfFileTask(child);                  task.fork();                  tasks.add(task);              }          }            long size = 0;          for (final SizeOfFileTask task : tasks) {              size += task.join();          }            return size;      }  } |

Tạo một chương trình sử dụng lớp tính kích thước ở trên:

* Khởi tạo ForkJoinPool để quản lý các RecursiveTask.
* Thực thi các task RecursiveTask trong ForkJoinPool.
* Hiển thị thông tin xử lý của ForkJoinPool sau mỗi 5 giây bao gồm: số lượng bộ xử lý (processor), số lượng Thread đang thực thi, số task chờ thực thi trong queue, số lượng task đã thực thi.
* Nhận kết quả xử lý từ RecursiveTask thông qua phương thức join().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | package com.gpcoder.threadpool.forkjoin;    import java.util.concurrent.ForkJoinPool;  import java.util.concurrent.TimeUnit;    public class RecursiveTaskExample {      public static void main(String[] args) {          // Create ForkJoinPool using the default constructor.          ForkJoinPool pool = new ForkJoinPool();            // Create three FolderProcessor tasks. Initialize each one with a different          // folder path.          SizeOfFileTask system = new SizeOfFileTask("C:/Windows");          SizeOfFileTask apps = new SizeOfFileTask("C:/Program Files");          SizeOfFileTask documents = new SizeOfFileTask("C:/Documents And Settings");            // Execute the three tasks in the pool using the execute() method.          pool.execute(system);          pool.execute(apps);          pool.execute(documents);            // Write to the console information about the status of the pool every second          // until the three tasks have finished their execution.          do {              System.out.printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");              System.out.printf("Main: Parallelism: %d\n", pool.getParallelism());              System.out.printf("Main: Active Threads: %d\n", pool.getActiveThreadCount());              System.out.printf("Main: Task Count: %d\n", pool.getQueuedTaskCount());              System.out.printf("Main: Steal Count: %d\n", pool.getStealCount());              System.out.printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");              try {                  TimeUnit.SECONDS.sleep(5);              } catch (InterruptedException e) {                  e.printStackTrace();              }          } while ((!system.isDone()) || (!apps.isDone()) || (!documents.isDone()));            // Shut down ForkJoinPool using the shutdown() method.          pool.shutdown();            // Write the number of results generated by each task to the console.          System.out.printf("Size of Windows: %d bytes \n", system.join());          System.out.printf("Size of Apps: %d bytes \n", apps.join());          System.out.printf("Size of Documents: %d bytes \n", documents.join());      }  } |

Thực thi chương trình trên, ta kết quả sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Main: Parallelism: 4  Main: Active Threads: 4  Main: Task Count: 285  Main: Steal Count: 5  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Main: Parallelism: 4  Main: Active Threads: 9  Main: Task Count: 3492  Main: Steal Count: 252  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Main: Parallelism: 4  Main: Active Threads: 70  Main: Task Count: 21958  Main: Steal Count: 23195  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Main: Parallelism: 4  Main: Active Threads: 96  Main: Task Count: 1202  Main: Steal Count: 60580  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Size of Windows: 24315219746 bytes  Size of Apps: 15642418058 bytes  Size of Documents: 0 bytes |

**Ví dụ sử dụng RecursiveAction**

Cú pháp khai báo một RecursiveAction như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class RecursiveTaskImpl extends RecursiveAction {      @Override      protected void compute() {            }  } |

Ý tưởng thực hiện của RecursiveAction hoàn toàn tương tự như RecursiveTask<T>, ngoại trừ phương thức compute() không trả kết quả xử lý về mà chỉ thực hiện công việc cho đến khi hoàn thành. Do đó, để có thể tính toán kích thước của folder, tôi sử dụng một biến trung gian để lưu trữ kết quả tính toán (AtomicLong sizeAccumulator).

[**AtomicLong**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicLong.html) là một kiểu dữ liệu tương tự như kiểu Long, nó được thiết kế để sử dụng trong các ứng dụng đa luồng.

**Atomatic** **Operation** (hoạt động nguyên tử) là một hoạt động hay một thao tác tính toán mà trong quá trình đó một processor có thể đồng bộ việc đọc và ghi dữ liệu trên cùng một bus tính toán. Điều này ngăn cản các processor khác hoặc thiết bị I/O từ việc ghi hay đọc bộ nhớ cho đến khi hoạt động atomic được hoàn tất.

Một hoạt động nguyên tử phải được thực hiện hoàn toàn hoặc không gì cả.

**AtomicLong** được thiết kế bằng cách sử dụng cơ chế đồng bộ để đảm bảo rằng các hoạt động get() từ bất kỳ thread khác là một thao tác đơn duy nhất. Có nghĩa là bất kỳ một thread khác, một khi thao tác được thực hiện nguyên tử, hoặc là sẽ thấy được giá trị trước khi được gán, hoặc là sau khi được gán. Tức là không bao giờ thấy được giá trị trung gian.

Một cách làm đơn giản việc này là khai báo biến đó kiểu sẽ được sửa đổi bới các threads khác nhau với từ khóa **volatile**. Do đó, biến volatile chỉ có 1 đối tượng nằm trên bộ nhớ. Tất cả các Thread khác đều có thể thấy được giá trị của nó, sau khi giá trị này được thay đổi kể cả ở khác thread.

Còn một cách khác nữa để đảm bảo hoạt động nguyên tử là sử dụng **synchronized** với phương thức **get()** và **set()** giá trị.