# 并行数据处理与性能

## 从串行到并行的性能测试

Stream 接口可以让你非常方便地处理它的元素：可以通过对收集源调用 parallelStream 方法来把集合转换为并行流。并行流就是一个把内容分成多个数据块，并用不同的线程分别处理每个数据块的流。这样一来，你就可以自动把给定操作的工作负荷分配给多核处理器的所有内核，让它们都忙起来。

让我们用一个简单的例子来试验一下这个思想。假设你需要写一个方法，接受数字n作为参数，并返回从1到给定参数的所有数字的和。我们使用下面这个方法来返回执行时间：

*/\*\*  
 \* measureSumPerf方法接受一个函数和一个 long 作为参数,它会对传给方法的 long 应用函数10次，  
 \* 记录每次执行的时间（以毫秒为单位），并返回最短的一次执行时间，用这个框架来测试顺序加法器函数对前一千万个自然数求和要用多久。  
 \** ***@param adder*** *\** ***@param n*** *\** ***@return*** *\*/* **public static long** measureSumPerf(Function<Long, Long> adder, **long** n) {  
 **long** fastest = Long.***MAX\_VALUE***;  
 **for** (**int** i = 0; i < 10; i++) {  
 **long** start = System.*nanoTime*();  
 **long** sum = adder.apply(n);  
 **long** duration = (System.*nanoTime*() - start) / 1000000;  
*// System.out.println("Result: " + sum);* **if** (duration < fastest) fastest = duration;  
 }  
 **return** fastest;  
 }

### 数据结构方面的优化

一个直接的方法是生成一个无穷大的数字流，把它限制到给定的数目，然后用对两个数字求和的 BinaryOperator 来归约这个流，如下所示：

*// 方案一：***public static long** sequentialSum1(**long** n) {  
 **return** Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .reduce(0L, Long::*sum*);  
}

执行结果：

sequentialSum1:94 msecs

仔细思考这段代码会发现，该方法存在一个性能问题，就是它需要对生成的原始类型的数字做装箱和拆箱操作，我们可以使用数值流 LongStream 对此进行改进，如下：

*// 方案二：使用数值流，杜绝拆/装箱的额外开销***public static long** sequentialSum2(**long** n) {  
 **return** LongStream.*iterate*(1L, i -> i+1)  
 .limit(n)  
 .reduce(0L, Long::*sum*);  
}

执行结果：

sequentialSum2:17 msecs

另外，我们还可以使用传统 for 循环的迭代版本执行起来比方案二会快一点点，因为它更为底层，同时也不需要对原始类型做任何装箱或拆箱操作

*// 方案三：用传统 for 循环的迭代版本执行起来比方案二会快一点点，因为它更为底层，同时也不需要对原始类型做任何装箱或拆箱操作***public static long** sequentialSum3(**long** n) {  
 **long** result = 0;  
 **for** (**long** i = 1L; i <= n; i++) {  
 result += i;  
 }  
 **return** result;  
}

执行结果：

sequentialSum3:5 msecs

### 并行化处理

接下来我们尝试使用并行化进行优化，首先我们使用parallel()方法对流进行并行化操作，如：

*// 方案四：并行化处理***public static long** parallelSum1(**long** n) {  
 **return** Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .parallel()  
 .reduce(0L, Long::*sum*);  
}

执行结果：

parallelSum1:121 msecs

测试完该代码后，你会发现这个求和方法的并行版本比顺序版本（方案一）要慢很多。这里实际上有两个问题：

首先，iterate 生成的是装箱的对象，必须拆箱成数字才能求和，我们使用数值流LongStream对此进行优化，如：

*// 方案五：并行化，并杜绝拆装箱开销***public static long** parallelSum2(**long** n) {  
 **return** LongStream.*iterate*(1L, i -> i+1)  
 .limit(n)  
 .parallel()  
 .reduce(0L, Long::*sum*);  
}

执行结果：

parallelSum2:102 msecs

测试该代码，我们发现，优化程度不尽人意，其原因是我们很难把 iterate 分成多个独立块来并行执行，因为每次执行这个函数生成下一个自然数都要依赖前一次应用的结果，也就是必须等待执行完累加操作后才能在生成自然数。而流标记成并行，其实是给顺序处理增加了开销，它还要把每次求和操作分到一个不同的线程上。

那到底要怎么利用多核处理器，用流来高效地并行求和呢？我们可以使用LongStream.rangeClosed 的方法，这个方法与 Stream.iterate 相比有两个优点：

LongStream.rangeClosed 直接产生原始类型的 long 数字，没有装箱拆箱的开销。

LongStream.rangeClosed 会生成数字范围，很容易拆分为独立的小块。例如，范围1~20可分为1~5、6~10、11~15和16~20。

让我们先看一下LongStream.rangeClosed用于顺序流时的性能如何，看看拆箱的开销到底要不要紧：

*// 方案六：迭代问题的优化***public static long** RangedSum(**long** n) {  
 **return** LongStream.*rangeClosed*(1, n)  
 .reduce(0L, Long::*sum*);  
}

执行结果：

RangedSum:23 msecs

我们发现执行结果快了不少，因为数值流避免了非针对性流那些没必要的自动装箱和拆箱操作。由此可见，选择适当的数据结构往往比并行化算法更重要。

现在我们在加上并行化处理：

*// 方案七：迭代问题的优化并做并行处理***public static long** parallelRangedSum(**long** n) {  
 **return** LongStream.*rangeClosed*(1, n)  
 .parallel()  
 .reduce(0L, Long::*sum*);  
}

执行结果：

parallelRangedSum:6 msecs

终于，我们得到了一个比顺序执行更快的并行归纳。

### 完整的测试代码

**import** sun.rmi.runtime.Log;  
  
**import** java.util.Iterator;  
**import** java.util.function.Function;  
**import** java.util.function.Supplier;  
**import** java.util.logging.Logger;  
**import** java.util.stream.LongStream;  
**import** java.util.stream.Stream;  
  
*/\*\*  
 \* Created by wb-whz291815 on 2017/6/27.  
 \*/***public class** Demo {  
  
 *// 方案一：* **public static long** sequentialSum1(**long** n) {  
 **return** Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .reduce(0L, Long::*sum*);  
 }  
  
 *// 方案二：使用数值流，杜绝拆/装箱的额外开销* **public static long** sequentialSum2(**long** n) {  
 **return** LongStream.*iterate*(1L, i -> i+1)  
 .limit(n)  
 .reduce(0L, Long::*sum*);  
 }  
  
 *// 方案三：用传统 for 循环的迭代版本执行起来比方案二会快一点点，因为它更为底层，同时也不需要对原始类型做任何装箱或拆箱操作* **public static long** sequentialSum3(**long** n) {  
 **long** result = 0;  
 **for** (**long** i = 1L; i <= n; i++) {  
 result += i;  
 }  
 **return** result;  
 }  
  
 *// 方案四：并行化处理* **public static long** parallelSum1(**long** n) {  
 **return** Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .parallel()  
 .reduce(0L, Long::*sum*);  
 }  
  
 *// 方案五：并行化，并杜绝拆装箱开销* **public static long** parallelSum2(**long** n) {  
 **return** LongStream.*iterate*(1L, i -> i+1)  
 .limit(n)  
 .parallel()  
 .reduce(0L, Long::*sum*);  
 }  
  
 *// 方案六：迭代问题的优化* **public static long** RangedSum(**long** n) {  
 **return** LongStream.*rangeClosed*(1, n)  
 .reduce(0L, Long::*sum*);  
 }  
  
 *// 方案七：迭代问题的优化并做并行处理* **public static long** parallelRangedSum(**long** n) {  
 **return** LongStream.*rangeClosed*(1, n)  
 .parallel()  
 .reduce(0L, Long::*sum*);  
 }  
  
 */\*\*  
 \* measureSumPerf方法接受一个函数和一个 long 作为参数,它会对传给方法的 long 应用函数10次，  
 \* 记录每次执行的时间（以毫秒为单位），并返回最短的一次执行时间，用这个框架来测试顺序加法器函数对前一千万个自然数求和要用多久。  
 \** ***@param adder*** *\** ***@param n*** *\** ***@return*** *\*/* **public static long** measureSumPerf(Function<Long, Long> adder, **long** n) {  
 **long** fastest = Long.***MAX\_VALUE***;  
 **for** (**int** i = 0; i < 10; i++) {  
 **long** start = System.*nanoTime*();  
 **long** sum = adder.apply(n);  
 **long** duration = (System.*nanoTime*() - start) / 1000000;  
*// System.out.println("Result: " + sum);* **if** (duration < fastest) fastest = duration;  
 }  
 **return** fastest;  
 }  
  
 **public static void** main(String[] args) {  
 *//System.out.println("本机器的处理器数量：" + Runtime.getRuntime().availableProcessors());* System.***out***.println(**"sequentialSum1:"** + *measureSumPerf*(Demo::*sequentialSum1*, 10000000) + **" msecs"**);*//sequentialSum:96 msecs* System.***out***.println(**"sequentialSum2:"** + *measureSumPerf*(Demo::*sequentialSum2*, 10000000) + **" msecs"**);*//sequentialSum:96 msecs* System.***out***.println(**"sequentialSum3:"** + *measureSumPerf*(Demo::*sequentialSum3*, 10000000) + **" msecs"**);*//sequentialSum:96 msecs* System.***out***.println(**"parallelSum1:"** + *measureSumPerf*(Demo::*parallelSum1*, 10000000) + **" msecs"**);*//Iterative:6 msecs* System.***out***.println(**"parallelSum2:"** + *measureSumPerf*(Demo::*parallelSum2*, 10000000) + **" msecs"**);*//parallelSum:134 msec* System.***out***.println(**"RangedSum:"** + *measureSumPerf*(Demo::*RangedSum*, 10000000) + **" msecs"**);*//parallelRangedSum:1 msecs* System.***out***.println(**"parallelRangedSum:"** + *measureSumPerf*(Demo::*parallelRangedSum*, 10000000) + **" msecs"**);*//parallelRangedSum:1 msecs* }  
}

执行结果：

sequentialSum1:94 msecs

sequentialSum2:17 msecs

sequentialSum3:5 msecs

parallelSum1:121 msecs

parallelSum2:102 msecs

RangedSum:23 msecs

parallelRangedSum:6 msecs

### 流并行化的代价和建议

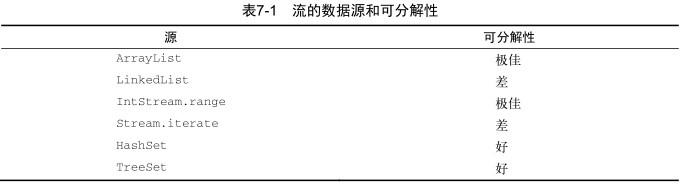
**并行化的代价：**

并行化并不是没有代价的。并行化过程本身需要对流做递归划分，把每个子流的归纳操作分配到不同的线程，然后把这些操作的结果合并成一个值。但在多个内核之间移动数据的代价也可能比你想的要大，所以很重要的一点是要保证在内核中并行执行工作的时间比在内核之间传输数据的时间长。总而言之，很多情况下不可能或不方便并行化。然而，在使用并行 Stream 加速代码之前，你必须确保用得对；如果结果错了，算得快就毫无意义了。

**建议：**

* 一般而言，想给出任何关于什么时候该用并行流的定量建议都是不可能也毫无意义的，因为任何类似于“仅当至少有一千个（或一百万个或随便什么数字）元素的时候才用并行流）”的建议对于某台特定机器上的某个特定操作可能是对的，但在略有差异的另一种情况下可能就是大错特错。尽管如此，我们至少可以提出一些定性意见，帮你决定某个特定情况下是否有必要使用并行流。
* 如果有疑问，测量。把顺序流转成并行流轻而易举，但却不一定是好事。我们在本节中已经指出，并行流并不总是比顺序流快。此外，并行流有时候会和你的直觉不一致，所以在考虑选择顺序流还是并行流时，第一个也是最重要的建议就是用适当的基准来检查其性能。
* 留意装箱。自动装箱和拆箱操作会大大降低性能。Java 8中有原始类型流（ IntStream 、LongStream 、 DoubleStream ）来避免这种操作，但凡有可能都应该用这些流。
* 有些操作本身在并行流上的性能就比顺序流差。特别是 limit 和 findFirst 等依赖于元素顺序的操作，它们在并行流上执行的代价非常大。例如， findAny 会比 findFirst 性能好，因为它不一定要按顺序来执行。你总是可以调用 unordered 方法来把有序流变成无序流。那么，如果你需要流中的n个元素而不是专门要前n个的话，对无序并行流调用limit 可能会比单个有序流（比如数据源是一个 List ）更高效。
* 还要考虑流的操作流水线的总计算成本。设N是要处理的元素的总数，Q是一个元素通过流水线的大致处理成本，则N\*Q就是这个对成本的一个粗略的定性估计。Q值较高就意味着使用并行流时性能好的可能性比较大。
* 对于较小的数据量，选择并行流几乎从来都不是一个好的决定。并行处理少数几个元素的好处还抵不上并行化造成的额外开销。
* 要考虑流背后的数据结构是否易于分解。例如， ArrayList 的拆分效率比 LinkedList高得多，因为前者用不着遍历就可以平均拆分，而后者则必须遍历。另外，用 range 工厂方法创建的原始类型流也可以快速分解。后面你将学到，自己实现Spliterator 来完全掌控分解过程。
* 流自身的特点，以及流水线中的中间操作修改流的方式，都可能会改变分解过程的性能。例如，一个 SIZED 流可以分成大小相等的两部分，这样每个部分都可以比较高效地并行处理，但筛选操作可能丢弃的元素个数却无法预测，导致流本身的大小未知。
* 还要考虑终端操作中合并步骤的代价是大是小（例如 Collector 中的 combiner 方法）。如果这一步代价很大，那么组合每个子流产生的部分结果所付出的代价就可能会超出通过并行流得到的性能提升。

表7-1按照可分解性总结了一些流数据源适不适于并行。



最后，我们还要强调并行流背后使用的基础架构是Java 7中引入的分支/合并框架。并行汇总的示例证明了要想正确使用并行流，了解它的内部原理至关重要，所以我们会在下一节仔细研究分支/合并框架。

# 分支/合并框架

分支/合并框架的目的是以递归方式将可以并行的任务拆分成更小的任务，然后将每个子任务的结果合并起来生成整体结果。它是 ExecutorService 接口的一个实现，它把子任务分配给线程池（称为 ForkJoinPool ）中的工作线程。首先来看看如何定义任务和子任务。要把任务提交到这个池，必须创建 RecursiveTask<R> 的一个子类，其中 R 是并行化任务产生的结果类型，或者如果任务不返回结果，则是 RecursiveAction 类型。要定义 RecursiveTask， 只需实现它唯一的抽象方法compute ：

protected abstract R compute();

这个方法同时定义了将任务拆分成子任务的逻辑，以及无法再拆分或不方便再拆分时，生成单个子任务结果的逻辑。正由于此，这个方法的实现类似于下面的伪代码：

if (任务足够小或不可分) {

顺序计算该任务

} else {

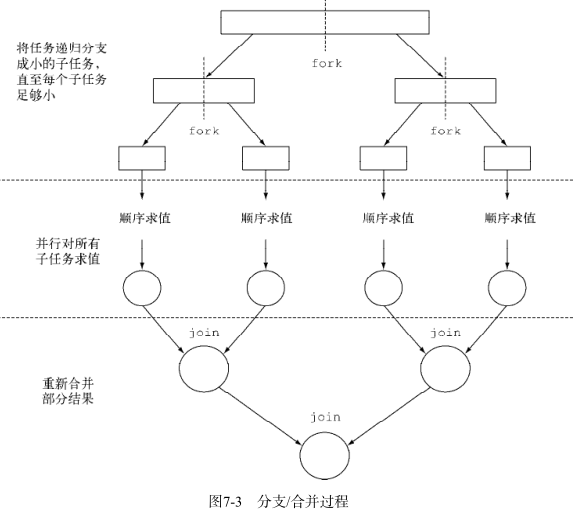
将任务分成两个子任务

递归调用本方法，拆分每个子任务，等待所有子任务完成

合并每个子任务的结果

}

递归的任务拆分过程如下图所示：



你可能已经注意到，这只不过是著名的分治算法的并行版本而已。

## 示例

这里举一个用分支/合并框架的实际例子，还以前面的例子为基础，让我们试着用这个框架为一个数字范围（这里用一个long[] 数组表示）求和。如前所述，你需要先为 RecursiveTask 类做一个实现，就是下面代码清单中的 ForkJoinSumCalculator 。

**import** java.util.concurrent.ForkJoinPool;  
**import** java.util.concurrent.ForkJoinTask;  
**import** java.util.function.Function;  
**import** java.util.stream.LongStream;  
  
**public class** ForkJoinSumCalculator **extends** java.util.concurrent.RecursiveTask<Long> {  
  
 **private final long**[] **numbers**;  
 **private final int start**;  
 **private final int end**;  
 **public static final long *THRESHOLD*** = 10\_000;  
  
 **public** ForkJoinSumCalculator(**long**[] numbers) {  
 **this**(numbers, 0, numbers.**length**);  
 }  
 **private** ForkJoinSumCalculator(**long**[] numbers, **int** start, **int** end) {  
 **this**.**numbers** = numbers;  
 **this**.**start** = start;  
 **this**.**end** = end;  
 }  
  
 @Override  
 **protected** Long compute() {  
 *// 该任务负责求和的部分的大小* **int** length = **end** - **start**;  
 **if** (length <= ***THRESHOLD***) {  
 **return** computeSequentially();  
 }  
 ForkJoinSumCalculator leftTask = **new** ForkJoinSumCalculator(**numbers**, **start**, **start** + length / 2);  
 *// 利用另一个 ForkJoinPool 线程异步执行新创建的子任务* leftTask.fork();  
  
 ForkJoinSumCalculator rightTask = **new** ForkJoinSumCalculator(**numbers**, **start** + length / 2, **end**);  
 *// 同步执行第二个子任务，有可能允许进一步递归划分* Long rightResult = rightTask.compute();  
  
 *// 读取第一个子任务的结果，如果尚未完成就等待* Long leftResult = leftTask.join(); *// 该任务的结果是两个子任务结果的组合* **return** leftResult + rightResult;  
 }  
  
 *// 子任务不再可分时计算结果的简单算法* **private long** computeSequentially() {  
 **long** sum = 0;  
 **for** (**int** i = **start**; i < **end**; i++) {  
 sum += **numbers**[i];  
 }  
 **return** sum;  
 }  
  
 *// 累加器* **public static long** forkJoinSum(**long** n) {  
 **long**[] numbers = LongStream.*rangeClosed*(1, n).toArray();  
 ForkJoinTask<Long> task = **new** ForkJoinSumCalculator(numbers);  
  
 **return new** ForkJoinPool().invoke(task);  
 }  
  
 *// 返回方法的执行时间* **public static long** measureSumPerf(Function<Long, Long> adder, **long** n) {  
 **long** fastest = Long.***MAX\_VALUE***;  
 **for** (**int** i = 0; i < 10; i++) {  
 **long** start = System.*nanoTime*();  
 **long** sum = adder.apply(n);  
 System.***out***.println(sum);  
 **long** duration = (System.*nanoTime*() - start) / 1\_000\_000;  
 **if** (duration < fastest) fastest = duration;  
 }  
 **return** fastest;  
 }  
  
 **public static void** main(String[] args) {  
 System.***out***.println(**"ForkJoin sum done in: "** +  
 *measureSumPerf*(ForkJoinSumCalculator::*forkJoinSum*, 10\_000\_000) + **" msecs"** );  
 }  
}

执行结果：

ForkJoin sum done in: 43 msecs

这个性能看起来比用并行流的版本要差，但这只是因为必须先要把整个数字流都放进一个long[] ，之后才能在 ForkJoinSumCalculator 任务中使用它。

## 使用分支/合并框架的注意事项

虽然分支/合并框架还算简单易用，不幸的是它也很容易被误用。以下是几个有效使用它的最佳做法。

* 对一个任务调用 join 方法会阻塞调用方，直到该任务做出结果。因此，有必要在两个子任务的计算都开始之后再调用它。否则，你得到的版本会比原始的顺序算法更慢更复杂，因为每个子任务都必须等待另一个子任务完成才能启动。
* 不应该在 RecursiveTask 内部使用 ForkJoinPool 的 invoke 方法。相反，你应该始终直接调用 compute 或 fork 方法，只有顺序代码才应该用 invoke 来启动并行计算。
* 对子任务调用 fork 方法可以把它排进 ForkJoinPool 。同时对左边和右边的子任务调用它似乎很自然，但这样做的效率要比直接对其中一个调用 compute 低。这样做你可以为其中一个子任务重用同一线程，从而避免在线程池中多分配一个任务造成的开销。
* 调试使用分支/合并框架的并行计算可能有点棘手。特别是你平常都在你喜欢的IDE里面看栈跟踪（stack trace）来找问题，但放在分支合并计算上就不行了，因为调用 compute的线程并不是概念上的调用方，后者是调用 fork 的那个。
* 和并行流一样，你不应理所当然地认为在多核处理器上使用分支/合并框架就比顺序计算快。我们已经说过，一个任务可以分解成多个独立的子任务，才能让性能在并行化时有所提升。所有这些子任务的运行时间都应该比分出新任务所花的时间长；一个惯用方法是把输入/输出放在一个子任务里，计算放在另一个里，这样计算就可以和输入/输出同时进行。此外，在比较同一算法的顺序和并行版本的性能时还有别的因素要考虑。就像任何其他Java代码一样，分支/合并框架需要“预热”或者说要执行几遍才会被JIT编译器优化。这就是为什么在测量性能之前跑几遍程序很重要，我们的测试框架就是这么做的。同时还要知道，编译器内置的优化可能会为顺序版本带来一些优势（例如执行死码分析——删去从未被使用的计算）。对于分支/合并拆分策略还有最后一点补充：你必须选择一个标准，来决定任务是要进一步拆分还是已小到可以顺序求值。

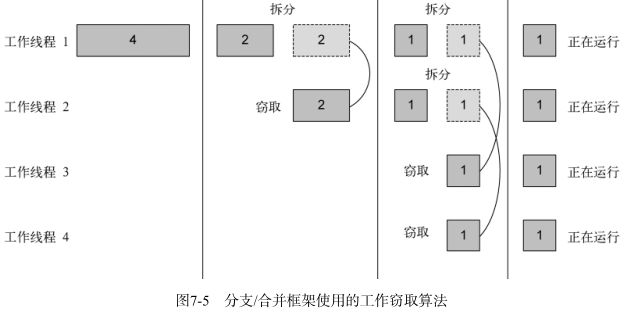
## 工作窃取

在 ForkJoinSumCalculator 的例子中，我们决定在要求和的数组中最多包含10 000个项目时就不再创建子任务了。这个选择是很随意的，但大多数情况下也很难找到一个好的启发式方法来确定它，只能试几个不同的值来尝试优化它。在我们的测试案例中，我们先用了一个有1000万项目的数组，意味着 ForkJoinSumCalculator 至少会分出1000个子任务来。这似乎有点浪费资源，因为我们用来运行它的机器上只有四个内核。在这个特定例子中可能确实是这样，因为所有的任务都受CPU约束，预计所花的时间也差不多。

但分出大量的小任务一般来说都是一个好的选择。这是因为，理想情况下，划分并行任务时，应该让每个任务都用完全相同的时间完成，让所有的CPU内核都同样繁忙。不幸的是，实际中，每个子任务所花的时间可能天差地别，要么是因为划分策略效率低，要么是有不可预知的原因，比如磁盘访问慢，或是需要和外部服务协调执行。

分支/合并框架工程用一种称为工作窃取（work stealing）的技术来解决这个问题。在实际应用中，这意味着这些任务差不多被平均分配到 ForkJoinPool 中的所有线程上。每个线程都为分配给它的任务保存一个双向链式队列，每完成一个任务，就会从队列头上取出下一个任务开始执行。基于前面所述的原因，某个线程可能早早完成了分配给它的所有任务，也就是它的队列已经空了，而其他的线程还很忙。这时，这个线程并没有闲下来，而是随机选了一个别的线程，从队列的尾巴上“偷走”一个任务。这个过程一直继续下去，直到所有的任务都执行完毕，所有的队列都清空。这就是为什么要划成许多小任务而不是少数几个大任务，这有助于更好地在工作线程之间平衡负载。

一般来说，这种工作窃取算法用于在池中的工作线程之间重新分配和平衡任务。下图展示了这个过程。当工作线程队列中有一个任务被分成两个子任务时，一个子任务就被闲置的工作线程“偷走”了。如前所述，这个过程可以不断递归，直到规定子任务应顺序执行的条件为真。



## 可分迭代器：Spliterator

本节中我们分析了一个例子，你明确地指定了将数字数组拆分成多个任务的逻辑。但是，使用本章前面讲的并行流时就用不着这么做了，这就意味着，肯定有一种自动机制来为你拆分流。这种新的自动机制称为 Spliterator。

Spliterator 是Java 8中加入的另一个新接口；这个名字代表“可分迭代器”（splitableiterator）。和 Iterator 一样， Spliterator 也用于遍历数据源中的元素，但它是为了并行执行而设计的。虽然在实践中可能用不着自己开发 Spliterator ，但了解一下它的实现方式会让你对并行流的工作原理有更深入的了解。Java 8已经为集合框架中包含的所有数据结构提供了一个默认的 Spliterator 实现。集合实现了 Spliterator 接口，接口提供了一个 spliterator 方法。

这个接口定义了若干方法，如下面的代码清单所示。

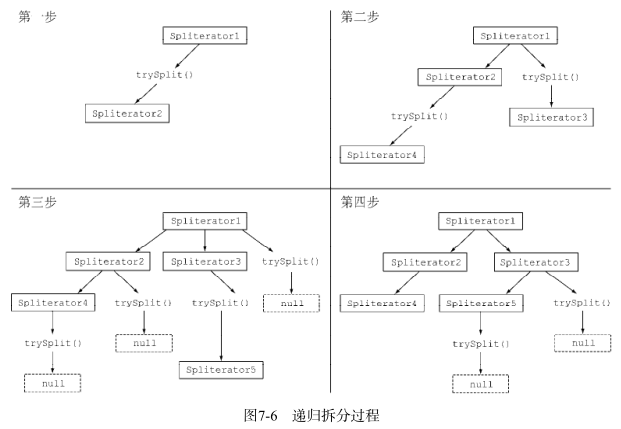
**public interface** Spliterator<T> {  
 **boolean** tryAdvance(Consumer<? **super** T> action);  
 Spliterator<T> trySplit();  
 **long** estimateSize();  
 **int** characteristics();  
}

与往常一样， T 是 Spliterator 遍历的元素的类型。 tryAdvance 方法的行为类似于普通的Iterator ，因为它会按顺序一个一个使用 Spliterator 中的元素，并且如果还有其他元素要遍历就返回 true 。但 trySplit 是专为 Spliterator 接口设计的，因为它可以把一些元素划出去分给第二个 Spliterator （由该方法返回），让它们两个并行处理。 Spliterator 还可通过estimateSize 方法估计还剩下多少元素要遍历，因为即使不那么确切，能快速算出来是一个值也有助于让拆分均匀一点。

重要的是，要了解这个拆分过程在内部是如何执行的，以便在需要时能够掌控它。因此，我们会在下一节中详细地分析它。

### 拆分过程

将 Stream 拆分成多个部分的算法是一个递归过程，如图7-6所示。第一步是对第一个Spliterator 调用 trySplit ，生成第二个 Spliterator 。第二步对这两个 Spliterator 调用trysplit ，这样总共就有了四个 Spliterator 。这个框架不断对 Spliterator 调用 trySplit直到它返回 null ，表明它处理的数据结构不能再分割，如第三步所示。最后，这个递归拆分过程到第四步就终止了，这时所有的 Spliterator 在调用 trySplit 时都返回了 null 。



这个拆分过程也受 Spliterator 本身的特性影响，而特性是通过 characteristics 方法声明的。

### Spliterator 的特性

Spliterator 接口声明的最后一个抽象方法是 characteristics ，它将返回一个 int ，代表 Spliterator 本身特性集的编码。使用 Spliterator 的客户可以用这些特性来更好地控制和优化它的使用。表7-2总结了这些特性。（不幸的是，虽然它们在概念上与收集器的特性有重叠，编码却不一样。）

