# 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

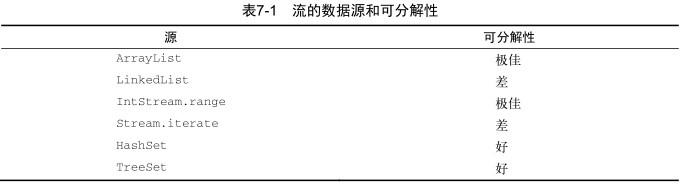
**并行化的代价：**

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。