链滴

【翻译】Java 8 中的并行流处理 -- 顺序流处理与并行流处理的性能对比

作者：MingGH

原文链接：https://ld246.com/article/1629726119415

来源网站：链滴

Java 8中的并行流处理--顺序流处理与并行流处理的性能比

出处：https://blog.oio.de/2016/01/22/parallel-stream-processing-in-java-8-performance-of-sequential-vs-parallel-stream-processing

并行处理在当今社会无处不在。由于cpu核心数量的增加和硬件成本的降低，使得集群系统更加便宜并行处理似乎是下一个big thing。

Java 8通过新的流API和在集合和数组上创建并行处理的简化来关注这一事实。让我们来看看这是如工作的。

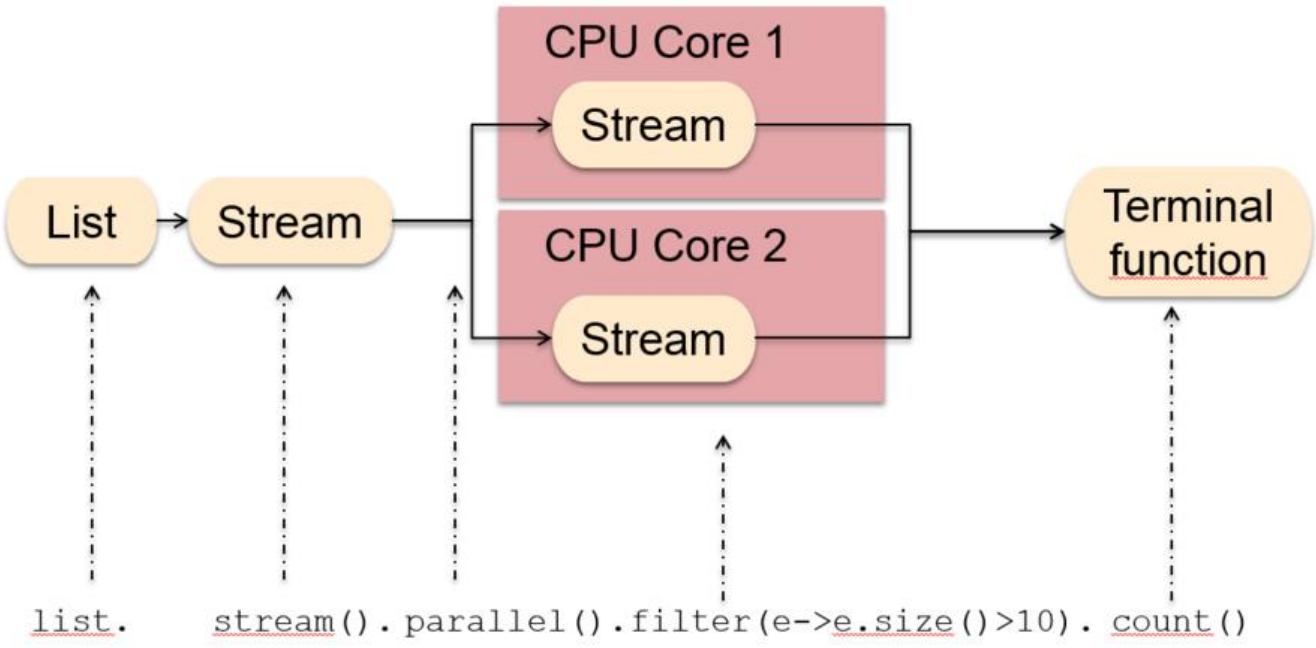
假设myList是一个整数列表，包含500.000个整数值。在前java 8时代，对这些整数值进行汇总的方是使用for each循环。

```
for (int i :myList)
result+=i;
```

从java 8开始，我们可以用流来做同样的事情

```
myList.stream().sum();
```

并行化处理非常容易，我们只需用关键字parallelStream()来代替stream，或者如果我们还有一个stream，就用parallel()。



所以

并行化流操作代码应该是这个样子的

```
myList.parallelStream().sum()
```

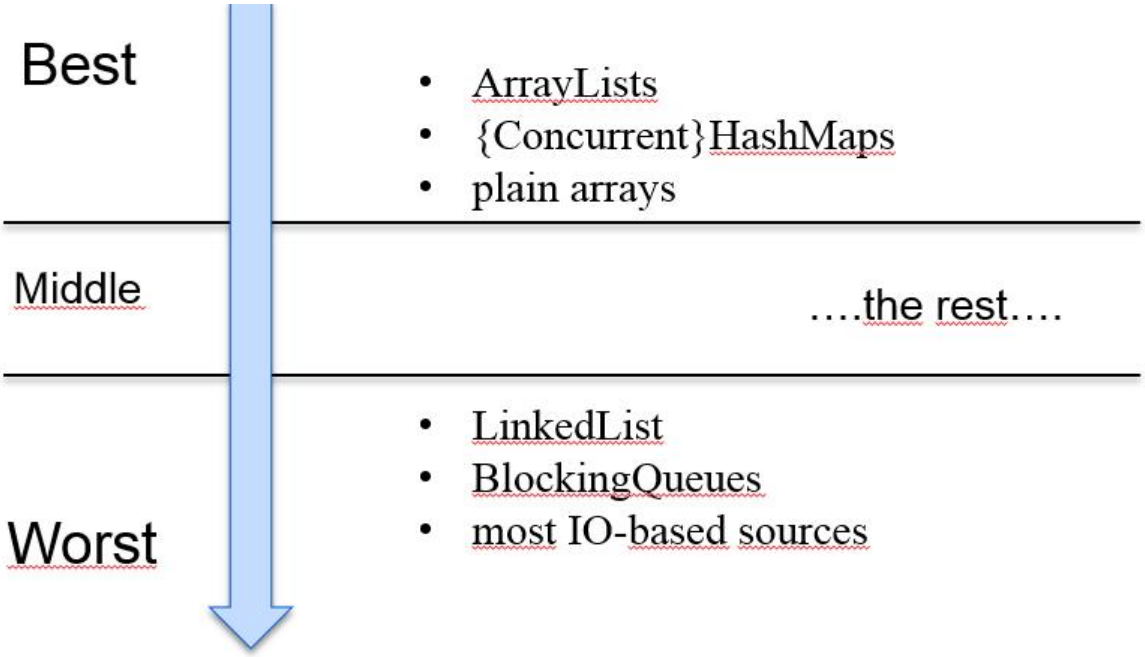
这样的改写很容易将计算分散到线程和可用的cpu核心上。但我们知道，多线程和并行处理的开销是贵的。问题是什么时候使用并行流，什么时候使用串行流会更有利于性能。

原文链接： 【翻译】Java 8 中的并行流处理 -- 顺序流处理与并行流处理的性能对比

首先让我们看一下幕后发生了什么。并行流使用Fork/Join框架进行处理。这意味着stream-source被forked(也就是被拆分)，并交给fork/join-pool的workers执行。

但在这里我们发现需要思考的第一点，并不是所有的stream-source被都可以像其他流源一样被分割想想ArrayList，它的内部数据表示是基于一个数组的。拆分这样的stream很容易，因为可以计算中元素的索引并拆分数组。

如果我们有一个LinkedList，那么分割数据元素就会更加复杂。实现者必须从第一个元素开始浏览所的元素，找到可以进行分割的元素。因此，例如LinkedLists对于并行流来说表现得很糟糕。



这是我们可以保留的第一个关于平行流性能的事实。

S - 源集合(collection)必须是可有效分割的

分割一个集合，管理fork和join任务，对象创建和垃圾收集也是一种算法上的开销。只有当需要在cp核上完成的工作非同小可和/或集合足够大时，这才是值得的。当然，我们也有很多cpu核。

一个错误的例子是计算5个整数值的最大值。

IntStream.rangeClosed(1, 5).reduce(Math::max).getAsInt();
里为fork/join准备和处理数据的开销是如此之大，以至于这里的串行流要快得多。Math.max函数在里的CPU成本不是很高，而且我们的数据元素较少

但是，当每个元素执行的函数更复杂时，它就越来越有价值了--确切地说，是 "更密集的cpu"。例如计算每个元素的正弦值而不是最大值。

当对国际象棋游戏进行编程时，每一步棋的评估也是这样的例子。许多评估可以并行进行。而且我们大量可能的下一步棋。

这对并行处理来说是完美的。

而这是我们可以保留的第二个关于平行流性能的事实。

NQ - "元素数量每个元素的成本 "的系数应该很大

但这也意味着反过来说，当每个元素的操作成本较高时，集合可以更小。

或者当每个元素的操作不是那么密集的时候，我们需要一个有很多元素的大集合，这样并行流的使用会有回报。

这直接取决于我们可以保留的第三个事实

C - CPU核心的数量 - 越多越好 > 1个是必须的

在单核机器上，由于管理开销的原因，并行流的表现总是比串行流差。就像公司有很多项目负责人而有一个人在做工作一样。

越多越好--不幸的是，在现实世界中，这并不是在所有情况下都是正确的，例如，当集合太小，CPU心启动时--也许是从energy safe mode--才发现没有什么可做的。

为了确定是否使用平行流，对每个元素的函数也有要求。这与其说是性能问题，不如说是并行流是否如期工作的问题。

该功能必须是...

- ...独立，这意味着每个元素的计算不能依赖或影响任何其他元素的计算。
- ...无干扰，这意味着函数在处理时不会修改基础数据源。
- ...无状态。

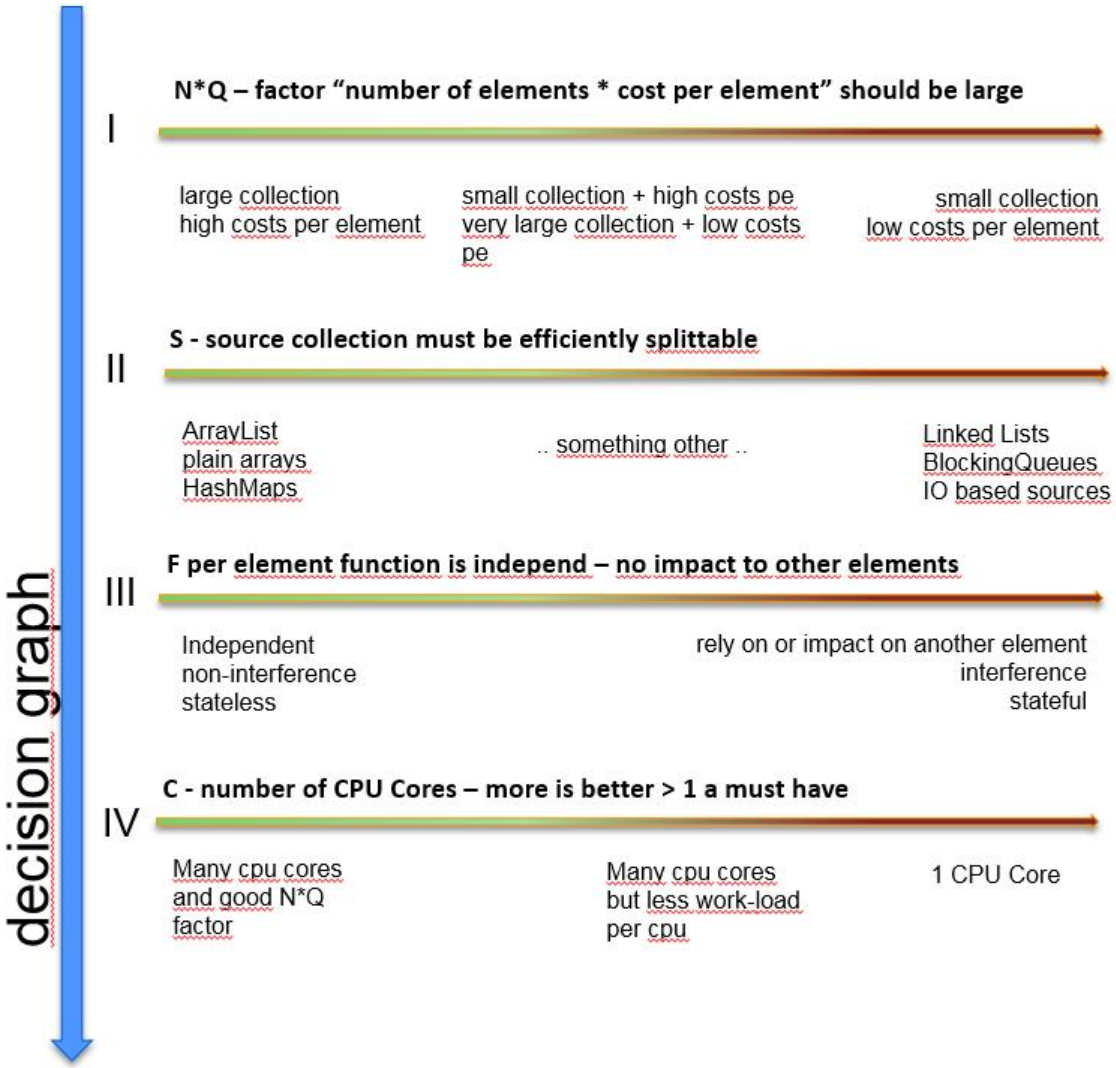
这里我们有一个在并行流中使用的有状态lambda函数的例子。这个例子取自java JDK API，显示了个简化的distinct()实现。

```
Set seen = Collections.synchronizedSet(new HashSet());
stream.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })...
```

因此，这让我们看到了我们可以保留的第四个事实。

F - 每个元素的函数必须是独立的

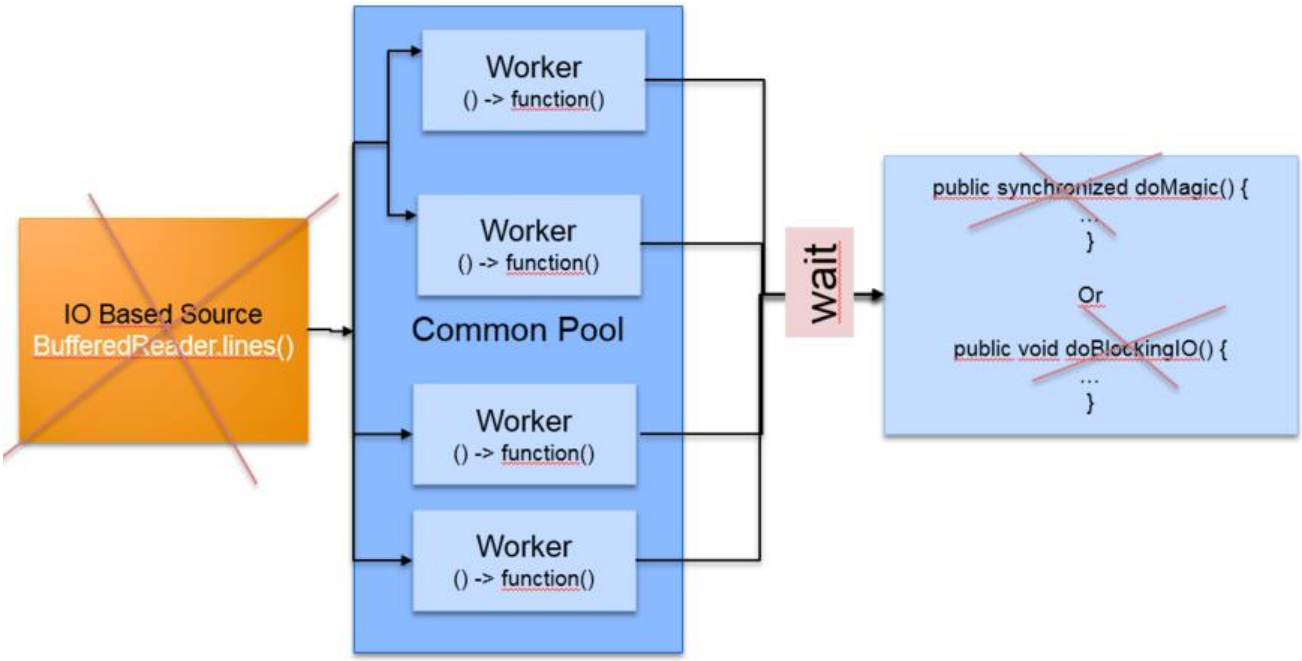
总结一下



还有其他一些情况下，我们不应该将我们的流并行化吗？是的，有的。

总是要考虑你的每个元素函数在做什么，以及这是否适合于并行处理的世界。当你的函数正在调用一同步功能时，那么你可能不会从并行化你的流中得到任何好处，因为你的并行流经常会在这个同步障上等待。

当你调用阻塞的i/o操作时也会出现同样的问题。



就这一点而言，使用基于I/O的源作为流也是众所周知的，因为数据是按顺序读取的，所以这样的源难被分割。