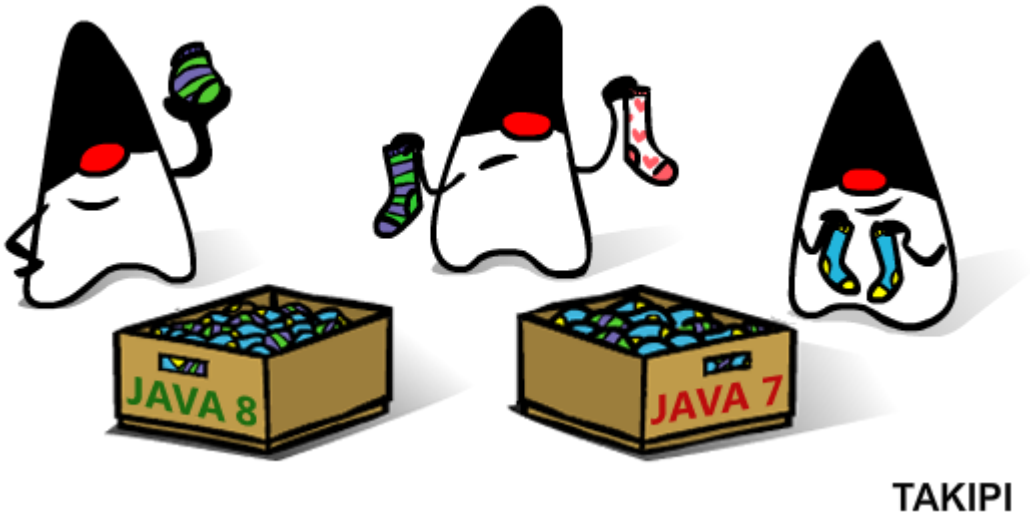
Java 8 的新并行 API – 魅力与炫目背后



Java 8 在多任务处理方面很优秀。让我们来看看它是怎么做的。

在 Java 8 引入的新功能中，有很重要的一项是并行数组处理。这项新功能使得我们能够使用可以利用多核体系结构的 Lambda 表达式来对对数组的元素进行排序，过滤和分组。这里的重点是，Java 程序员只需要非常少的工作就可以立刻使程序的性能获得提升。非常酷。

问题来了。这项新功能有多快？我应该什么时候使用它？好吧，答案有点让人沮丧——这依赖于具体的情况。要知道依赖什么情况吗？请继续阅读。

新的 API

Java8 的新并行操作 API 十分灵活。让我们一起看几个我们要用来做测试的例子。

1. 使用多核对数组进行**排序**：

```
1    Arrays.parallelSort(numbers);
```

2. 根据特定的条件（比如：素数和非素数）对数组进行**分组**：

```
1    Map<Boolean, List<Integer>> groupByPrimary = numbers
2       .parallelStream().collect(Collectors.groupingBy(s -> Utility.isPrime(s)));
```

3. 对数组进行**过滤**：

```
1    Integer[]   prims = numbers.parallelStream().filter(s -> Utility.isPrime(s))
2       .toArray();
```

跟自己写多线程程序来实现相同的功能比较，生产力提高太多了！在这个新的体系中，我个人最喜欢的是一个叫 Spliterator 的新概念，将一个集合分成多个块，并行处理这多个块并将处理结果汇合到一起。就像它的哥哥 iterator，它也被用来遍历一个集合的元素，只不过它更加灵活，允许你编写检查和分离集合的自定义行为，并在遍历时直接插入。

它的性能如何？

为了测试这些并行操作 API 的性能， 我在两种情况（低竞争和高竞争）下进行了实验。原因是单独运行一个多核算法，往往会有好的性能，但在真实的服务器环境中运行，情况就完全不同了。真实环境中往往有大量的线程在竞争宝贵的 CPU 时间片以处理消息或用户请求，由于竞争的存在，程序的性能就降低了。所以我进行了接下来的测试。我首先随机生成了长度为 100K 的整数数组，这些整数的取值在 0 到 1 百万之间。然后我分别使用传统的顺序方法和新的 Java 8 的并行 API 对这个数组进行了排序，分组和过滤。结果并不使人惊讶。

- **快速排序**快了 **4.7** 倍
- **分组**快了 **5** 倍

- **过滤**快了 **5.5** 倍

这可以说明 java 8 的并行 API 具有非常好的性能吗？很不幸，不能。

Sort (ms)	Parallel Sort	Group (ms)	Parallel group	Filter (ms)	Parallel filter
23	4	18	4	18	3
20	4	15	3	16	3
20	5	17	3	17	3
20	4	18	3	29	3
20	5	15	3	15	3
19	5	17	3	15	3
21	4	16	3	17	3
19	4	16	4	14	3
19	4	17	3	18	3
19	4	15	3	15	3
200	43	164	32	174	30

*测试结果与运行了 100 次的附加测试结果一致。

*测试机器为 MBP，i7 四核。

在有负载的情况下会发生什么呢？

目前为止新 API 的性能表现非常出色，原因是线程之间对 CPU 的时间片的竞争非常少。这是理想的环境，但不幸的是，理想环境往往不会出现在现实环境中。为了模拟真实的环境，我建立了第二个测试。这次测试使用跟第一次相同的算法，

但测试任务在十个并发线程上执行 ,以模拟处在压力环境中的服务器同时处理十个请求的情况。这十个请求使用传统的顺利处理方法或 Java 8 的新 API 处理。

测试结果

- **排序**现在只快了 20%
- **过滤**现在只快了 20%
- **分组**现在满了 15%

更高的规模和竞争水平很可能使这些数字进一步下降。原因是在一个多线程的环境中添加线程并不一定能帮助你提高计算效率，是计算机的 CPU 个数决定了计算效率，而不是线程个数。

Sort (ms)	Parallel Sort	Group (ms)	Parallel group	Filter (ms)	Parallel filter
36	29	19	26	28	29
39	27	22	48	30	29
33	38	25	24	30	21
37	27	25	24	30	24
39	35	23	24	30	22
39	34	27	24	32	22
37	37	30	25	31	21
30	33	25	22	30	20
30	37	22	37	24	19
31	35	22	37	26	18
351	332	240	291	291	225

结论

虽然这些都是非常强大和易于使用的 API，但它们不是银弹。我们仍然需要花费精力去判断何时应该使用它们。如果你事先知道你会做多个处理并行操作，那么考虑使用排队架构，并使并发操作数和你的处理器数量相匹配可能是一个好主意。这里的难点在于运行时性能将依赖于实际的硬件体系结构和服务器所处的压力情况。你可能只有在压力测试或者生产环境中才能看到代码的运行时性能，使之成为一个“易编码，难调试”的经典案例。