[清单 1. Java 7 的排序、取值实现 10](#_Toc14777236)

[清单 2. Java 8 的排序、取值实现 10](#_Toc14777237)

[清单 3. 一个流操作的示例 11](#_Toc14777238)

[清单 4. 构造流的几种常见方法 12](#_Toc14777239)

[清单 5. 数据流的构造 13](#_Toc14777240)

[清单 6. 流转换为其他数据结构 13](#_Toc14777241)

[清单 7. 转换大写，这段代码把所有的单词转换为大写。 14](#_Toc14777242)

[清单 8. 平方数 15](#_Toc14777243)

[清单 9. 一对多 15](#_Toc14777244)

[清单 10. 留下偶数 16](#_Toc14777245)

[清单 11. 把单词挑出来，这段代码首先把每行的单词用 flatMap 整理到新的 Stream，然后保留长度不为 0 的，就是整篇文章中的全部单词了。 16](#_Toc14777246)

[清单 12. 打印姓名（forEach 和 pre-java8 的对比） 17](#_Toc14777247)

[清单 13. peek 对每个元素执行操作并返回一个新的 Stream 17](#_Toc14777248)

[清单 14. Optional 的两个用例 18](#_Toc14777249)

[清单 15. reduce 的用例 19](#_Toc14777250)

[清单 16. limit 和 skip 对运行次数的影响 20](#_Toc14777251)

[清单 17. limit 和 skip 对 sorted 后的运行次数无影响 21](#_Toc14777252)

[清单 18. 优化：排序前进行 limit 和 skip 21](#_Toc14777253)

[清单 19. 找出最长一行的长度 22](#_Toc14777254)

[清单 20. 找出全文的单词，转小写，并排序 22](#_Toc14777255)

[清单 21. 使用 Match 23](#_Toc14777256)

# Stream 作为 Java 8 的一大亮点，它与 java.io 包里的 InputStream 和 OutputStream 是完全不同的概念。它也不同于 StAX 对 XML 解析的 Stream，也不是 Amazon Kinesis 对大数据实时处理的 Stream。Java 8 中的 Stream 是对集合（Collection）对象功能的增强，它专注于对集合对象进行各种非常便利、高效的聚合操作（aggregate operation），或者大批量数据操作 (bulk data operation)，Stream API 借助于同样新出现的 Lambda 表达式，极大的提高编程效率和程序可读性。同时它提供串行和并行两种模式进行汇聚操作。

聚合操作函数一般有COUNT、SUM、MAX、MIN等函数

Stream 不是集合元素，它不是数据结构并不保存数据，它是有关算法和计算的，它**更像一个高级版本的 Iterator**。原始版本的 Iterator，用户只能显式地一个一个遍历元素并对其执行某些操作；高级版本的 Stream，用户只要给出需要**对其包含的元素执行什么操作**，比如 “过滤掉长度大于 10 的字符串”、“获取每个字符串的首字母”等，Stream 会**隐式**地在内部进行遍历，做出相应的数据转换。

Stream 就如同一个迭代器（Iterator），单向，不可往复，数据**只能遍历一次**，遍历过一次后即用尽了，就好比流水从面前流过，一去不复返。

而和迭代器又不同的是，Stream 可以**并行化操作**，迭代器只能命令式地、串行化操作。顾名思义，当使用串行方式去遍历时，每个 item 读完后再读下一个 item。而使用并行去遍历时，数据会被分成多个段，其中每一个都在不同的线程中处理，然后将结果一起输出。Stream 的并行操作依赖于 Java7 中引入的 Fork/Join 框架（JSR166y）来拆分任务和加速处理过程。Java 的并行 API 演变历程基本如下：

1. 1.0-1.4 中的 java.lang.Thread
2. 5.0 中的 java.util.concurrent
3. 6.0 中的 Phasers 等
4. 7.0 中的 Fork/Join 框架
5. 8.0 中的 Lambda

Stream 的另外一大特点是，**数据源本身可以是无限的**。

当我们使用一个流的时候，通常包括三个基本步骤：

**获取一个数据源（source）→ 数据转换（transform values） → 执行操作获取想要的结果（operations）**

每次转换原有 Stream 对象不改变，**返回一个新**的 Stream 对象（可以有多次转换），这就允许对其操作可以像链条一样排列，变成一个管道，如下图所示。

# 

生成stream的方式的几种方式

从 Collection 和数组

* + Collection.stream()
  + Collection.parallelStream()
  + Arrays.stream(T array) or Stream.of()

从 BufferedReader

* + java.io.BufferedReader.lines()

静态工厂

* java.util.stream.IntStream.range()
* java.nio.file.Files.walk()

自己构建

* + java.util.Spliterator

其它

* + Random.ints()
  + BitSet.stream()
  + Pattern.splitAsStream(java.lang.CharSequence)
  + JarFile.stream()

# 流操作包括以下两种，中间操作、结束操作、短路操作

* **Intermediate（中间操作）**：一个流可以后面跟随零个或多个 intermediate 操作。其目的主要是打开流，做出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用。这类操作都是**惰性化**的（lazy），就是说，仅仅调用到这类方法，并没有真正开始流的遍历。
* **Terminal（结束操作）**：一个流只能有一个 terminal 操作，当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。Terminal 操作的执行，才会真正开始流的遍历，并且会生成一个结果，或者一个 side effect。

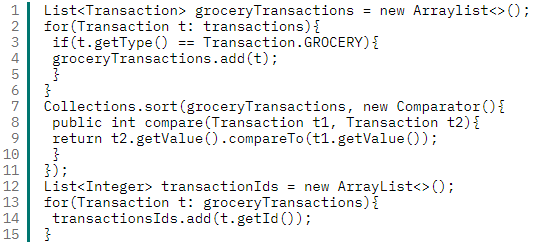
在对于一个 Stream 进行多次转换操作 (Intermediate 操作)，每次都对 Stream 的每个元素进行转换，而且是执行多次，这样时间复杂度就是 N（转换次数）个 for 循环里把所有操作都做掉的总和吗？其实不是这样的，转换操作都是 lazy 的，多个转换操作只会在 Terminal 操作的时候融合起来，一次循环完成。我们可以这样简单的理解，Stream 里有个操作函数的集合，每次转换操作就是把转换函数放入这个集合中，在 Terminal 操作的时候循环 Stream 对应的集合，然后对每个元素执行所有的函数。

还有一种操作被称为 **short-circuiting**。用以指：

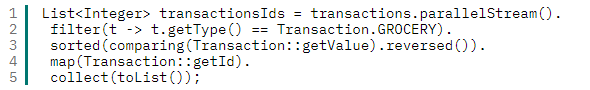
* 对于一个 intermediate 操作，如果它接受的是一个无限大（infinite/unbounded）的 Stream，但返回一个有限的新 Stream。
* 对于一个 terminal 操作，如果它接受的是一个无限大的 Stream，但能在有限的时间计算出结果。

当操作一个无限大的 Stream，而又希望在有限时间内完成操作，则在管道内拥有一个 short-circuiting 操作是必要非充分条件。

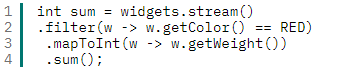
##### 清单 1. Java 7 的排序、取值实现



##### 清单 2. Java 8 的排序、取值实现



##### 清单 3. 一个流操作的示例

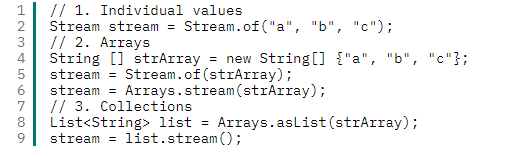


stream() 获取当前小物件的 source，filter 和 mapToInt 为 intermediate 操作，进行数据筛选和转换，最后一个 sum() 为 terminal 操作，对符合条件的全部小物件作重量求和。

使用详解

对 Stream 的使用就是实现一个 filter-map-reduce 过程，产生一个最终结果，或者导致一个副作用（side effect）。

##### 清单 4. 构造流的几种常见方法

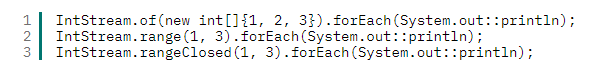


需要注意的是，对于基本数值型，目前有三种对应的包装类型 Stream：

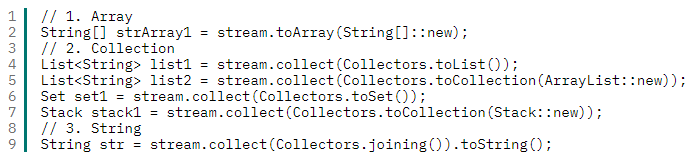
IntStream、LongStream、DoubleStream。当然我们也可以用 Stream<Integer>、Stream<Long> >、Stream<Double>，**但是 boxing 和 unboxing 会很耗时**，所以特别为这三种基本数值型提供了对应的 Stream。

Java 8 中还没有提供其它数值型 Stream，因为这将导致扩增的内容较多。而常规的数值型聚合运算可以通过上面三种 Stream 进行。

##### 清单 5. 数据流的构造



##### 清单 6. 流转换为其他数据结构



流的操作

* Intermediate：

map (mapToInt, flatMap 等)、 filter、 distinct、 sorted、 peek、 limit、 skip、 parallel、 sequential、 unordered

* Terminal：

forEach、 forEachOrdered、 toArray、 reduce、 collect、 min、 max、 count、 anyMatch、 allMatch、 noneMatch、 findFirst、 findAny、 iterator

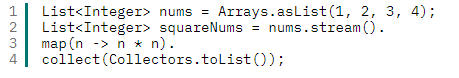
* Short-circuiting：

anyMatch、 allMatch、 noneMatch、 findFirst、 findAny、 limit

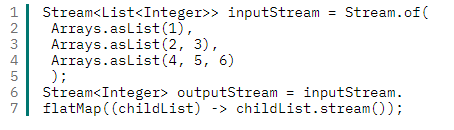
##### 清单 7. 转换大写，这段代码把所有的单词转换为大写。



##### 清单 8. 平方数



##### 清单 9. 一对多

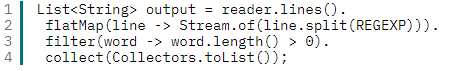


flatMap 把 input Stream 中的层级结构扁平化，就是将最底层元素抽出来放到一起，最终 output 的新 Stream 里面已经没有 List 了，都是直接的数字。

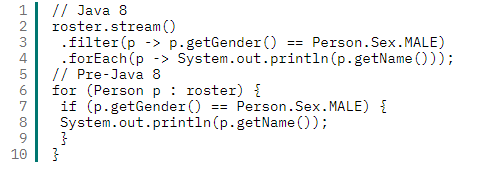
##### 清单 10. 留下偶数



##### 清单 11. 把单词挑出来，这段代码首先把每行的单词用 flatMap 整理到新的 Stream，然后保留长度不为 0 的，就是整篇文章中的全部单词了。

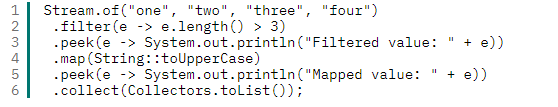


##### 清单 12. 打印姓名（forEach 和 pre-java8 的对比）

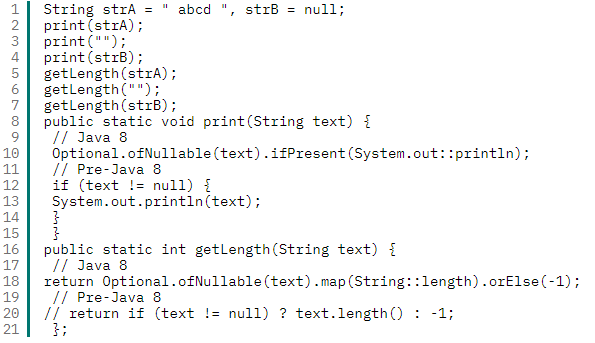


注意：forEach 方法接收一个 Lambda 表达式，然后在 Stream 的每一个元素上执行该表达式。forEach 不能修改自己包含的本地变量值，也不能用 break/return 之类的关键字提前结束循环。

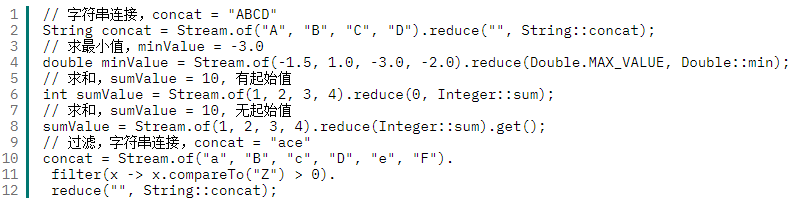
##### 清单 13. peek 对每个元素执行操作并返回一个新的 Stream



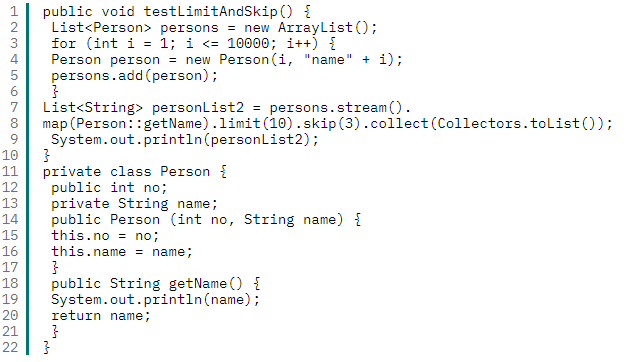
##### 清单 14. Optional 的两个用例



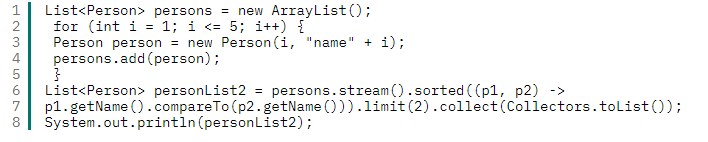
##### 清单 15. reduce 的用例



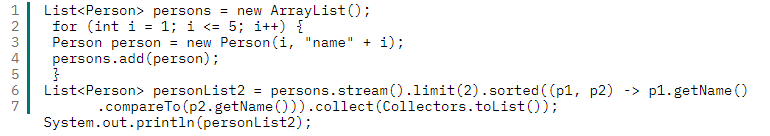
##### 清单 16. limit 和 skip 对运行次数的影响



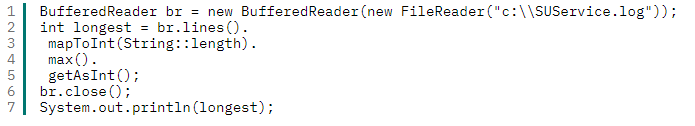
##### 清单 17. limit 和 skip 对 sorted 后的运行次数无影响



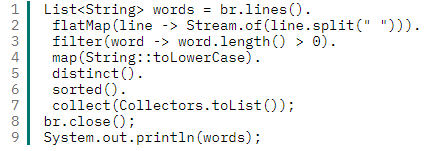
##### 清单 18. 优化：排序前进行 limit 和 skip



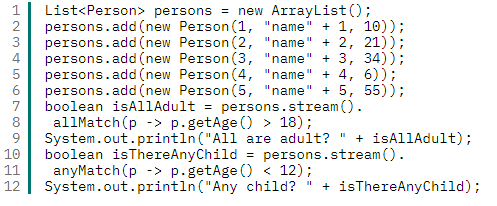
##### 清单 19. 找出最长一行的长度



##### 清单 20. 找出全文的单词，转小写，并排序



##### 清单 21. 使用 Match



Stream 有三个 match 方法，从语义上说：

* allMatch：Stream 中全部元素符合传入的 predicate，返回 true
* anyMatch：Stream 中只要有一个元素符合传入的 predicate，返回 true
* noneMatch：Stream 中没有一个元素符合传入的 predicate，返回 true

它们都不是要遍历全部元素才能返回结果。例如 allMatch 只要一个元素不满足条件，就 skip 剩下的所有元素，返回 false。对清单 13 中的 Person 类稍做修改，加入一个 age 属性和 getAge 方法。

## 结束语

总之，Stream 的特性可以归纳为：

* 不是数据结构
* 它没有内部存储，它只是用操作管道从 source（数据结构、数组、generator function、IO channel）抓取数据。
* 它也绝不修改自己所封装的底层数据结构的数据。例如 Stream 的 filter 操作会产生一个不包含被过滤元素的新 Stream，而不是从 source 删除那些元素。
* 所有 Stream 的操作必须以 lambda 表达式为参数
* 不支持索引访问
* 你可以请求第一个元素，但无法请求第二个，第三个，或最后一个。不过请参阅下一项。
* 很容易生成数组或者 List
* 惰性化
* 很多 Stream 操作是向后延迟的，一直到它弄清楚了最后需要多少数据才会开始。
* Intermediate 操作永远是惰性化的。
* 并行能力
* 当一个 Stream 是并行化的，就不需要再写多线程代码，所有对它的操作会自动并行进行的。
* 可以是无限的
  + 集合有固定大小，Stream 则不必。limit(n) 和 findFirst() 这类的 short-circuiting 操作可以对无限的 Stream 进行运算并很快完成

# [Java 8 中的 Streams API 详解](https://www.ibm.com/developerworks/cn/java/j-lo-java8streamapi/)

# [Java 8 Stream](https://www.runoob.com/java/java8-streams.html)