Java 8 中的 Streams API 详解

Streams 的背景，以及 Java 8 中的使用详解

陈 争云, 占 宇剑, 和 司 磊
2014 年 9 月 11 日发布 / *更新: 2019 年 5 月 05 日*

为什么需要 Stream

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，并发模式能够充分利用多核处理器的优势，使用 fork/join 并行方式来拆分任务和加速处理过程。通常编写并行代码很难而且容易出错, 但使用 Stream API 无需编写一行多线程的代码，就可以很方便地写出高性能的并发程序。所以说，Java 8 中首次出现的 java.util.stream 是一个函数式语言+多核时代综合影响的产物。

什么是聚合操作

在传统的 J2EE 应用中，Java 代码经常不得不依赖于关系型数据库的聚合操作来完成诸如：

- 客户每月平均消费金额
- 最昂贵的在售商品
- 本周完成的有效订单（排除了无效的）
- 取十个数据样本作为首页推荐

这类的操作。

但在当今这个数据大爆炸的时代，在数据来源多样化、数据海量化的今天，很多时候不得不脱离 RDBMS，或者以底层返回的数据为基础进行更上层的数据统计。而 Java 的集合 API 中，仅仅有极少量的辅助型方法，更多的时候是程序员需要用 Iterator 来遍历集合，完成相关的聚合应用逻辑。这是一种远不够高效、笨拙的方法。在 Java 7 中，如果要发现 type 为 grocery 的所有交易，然后返回以交易值降序排序好的交易 ID 集合，我们需要这样写：

清单 1. Java 7 的排序、取值实现

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
 if(t.getType() == Transaction.GROCERY){
 groceryTransactions.add(t);
 }
}
Collections.sort(groceryTransactions, new Comparator(){
 public int compare(Transaction t1, Transaction t2){
 return t2.getValue().compareTo(t1.getValue());
 }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
 transactionsIds.add(t.getId());
}
```

而在 Java 8 使用 Stream，代码更加简洁易读；而且使用并发模式，程序执行速度更快。

清单 2. Java 8 的排序、取值实现

```
List<Integer> transactionsIds = transactions.parallelStream().
 filter(t -> t.getType() == Transaction.GROCERY).
 sorted(comparing(Transaction::getValue).reversed()).
 map(Transaction::getId).
 collect(toList());
```

Stream 总览

什么是流

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

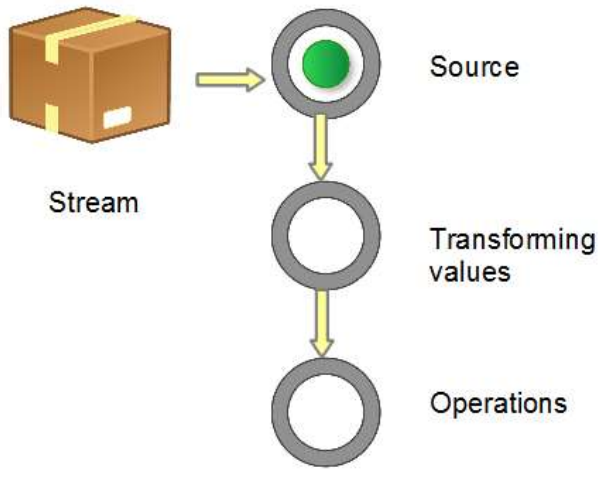
Stream 的另外一大特点是，数据源本身可以是无限的。

流的构成

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→执行操作获取想要的结果，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。

图 1. 流管道 (Stream Pipeline) 的构成



有多种方式生成 Stream Source：

- 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()

流的操作类型分为两种：

- **Intermediate**：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**：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 3. 一个流操作的示例

```
int sum = widgets.stream()
.filter(w -> w.getColor() == RED)
 .mapToInt(w -> w.getWeight())
 .sum();
```

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。

流的使用详解

简单说，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）。

流的构造与转换

下面提供最常见的几种构造 Stream 的样例。

清单 4. 构造流的几种常见方法

```
// 1. Individual values
Stream stream = Stream.of("a", "b", "c");
// 2. Arrays
String [] strArray = new String[] {"a", "b", "c"};
stream = Stream.of(strArray);
stream = Arrays.stream(strArray);
// 3. Collections
List<String> list = Arrays.asList(strArray);
stream = list.stream();
```

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. 数值流的构造

```
IntStream.of(new int[]{1, 2, 3}).forEach(System.out::println);
IntStream.range(1, 3).forEach(System.out::println);
IntStream.rangeClosed(1, 3).forEach(System.out::println);
```

清单 6. 流转换为其它数据结构

```
// 1. Array
String[] strArray1 = stream.toArray(String[]::new);
// 2. Collection
List<String> list1 = stream.collect(Collectors.toList());
List<String> list2 = stream.collect(Collectors.toCollection(ArrayList::new));
Set set1 = stream.collect(Collectors.toSet());
Stack stack1 = stream.collect(Collectors.toCollection(Stack::new));
// 3. String
String str = stream.collect(Collectors.joining()).toString();
```

一个 Stream 只可以使用一次，上面的代码为了简洁而重复使用了数次。

流的操作

接下来，当把一个数据结构包装成 Stream 后，就要开始对里面的元素进行各类操作了。常见的操作可以归类如下。

- 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

我们下面看一下 Stream 的比较典型用法。

map/flatMap

我们先来看 map。如果你熟悉 scala 这类函数式语言，对这个方法应该很了解，它的作用就是把 input Stream 的每一个元素，映射成 output Stream 的另外一个元素。

清单 7. 转换大写

```
List<String> output = wordList.stream().
map(String::toUpperCase).
collect(Collectors.toList());
```

这段代码把所有的单词转换为大写。

清单 8. 平方数

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4);
List<Integer> squareNums = nums.stream().
map(n -> n * n).
collect(Collectors.toList());
```

这段代码生成一个整数 list 的平方数 {1, 4, 9, 16}。

从上面例子可以看出，map 生成的是个 1:1 映射，每个输入元素，都按照规则转换成为另外一个元素。还有一些场景，是一对多映射关系的，这时需要 flatMap。

清单 9. 一对多

```
Stream<List<Integer>> inputStream = Stream.of(
 Arrays.asList(1),
 Arrays.asList(2, 3),
 Arrays.asList(4, 5, 6)
 );
Stream<Integer> outputStream = inputStream.
flatMap((childList) -> childList.stream());
```

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。

filter

filter 对原始 Stream 进行某项测试，通过测试的元素被留下来生成一个新 Stream。

清单 10. 留下偶数

```
Integer[] sixNums = {1, 2, 3, 4, 5, 6};
Integer[] evens =
Stream.of(sixNums).filter(n -> n%2 == 0).toArray(Integer[]::new);
```

经过条件“被 2 整除”的 filter，剩下的数字为 {2, 4, 6}。

清单 11. 把单词挑出来

```
List<String> output = reader.lines().
 flatMap(line -> Stream.of(line.split(REGEXP))).
 filter(word -> word.length() > 0).
 collect(Collectors.toList());
```

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。

forEach

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。

清单 12. 打印姓名（forEach 和 pre-java8 的对比）

```
// Java 8
roster.stream()
 .filter(p -> p.getGender() == Person.Sex.MALE)
 .forEach(p -> System.out.println(p.getName()));
// Pre-Java 8
for (Person p : roster) {
 if (p.getGender() == Person.Sex.MALE) {
 System.out.println(p.getName());
 }
}
```

对一个人员集合遍历，找出男性并打印姓名。可以看出来，forEach 是为 Lambda 而设计的，保持了最紧凑的风格。而且 Lambda 表达式本身是可以重用的，非常方便。当需要为多核系统优化时，可以 parallelStream().forEach()，只是此时原有元素的次序没法保证，并行的情况下将改变串行时操作的行为，此时 forEach 本身的实现不需要调整，而 Java8 以前的 for 循环 code 可能需要加入额外的多线程逻辑。

但一般认为，forEach 和常规 for 循环的差异不涉及到性能，它们仅仅是函数式风格与传统 Java 风格的差别。

另外一点需要注意，forEach 是 terminal 操作，因此它执行后，Stream 的元素就被“消费”掉了，你无法对一个 Stream 进行两次 terminal 运算。下面的代码是错误的：

```
stream.forEach(element -> doOneThing(element));
stream.forEach(element -> doAnotherThing(element));
```

相反，具有相似功能的 intermediate 操作 peek 可以达到上述目的。如下是出现在该 api javadoc 上的一个示例。

清单 13. peek 对每个元素执行操作并返回一个新的 Stream

```
Stream.of("one", "two", "three", "four")
 .filter(e -> e.length() > 3)
 .peek(e -> System.out.println("Filtered value: " + e))
 .map(String::toUpperCase)
 .peek(e -> System.out.println("Mapped value: " + e))
 .collect(Collectors.toList());
```

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。

findFirst

这是一个 termimal 兼 short-circuiting 操作，它总是返回 Stream 的第一个元素，或者空。

这里比较重点的是它的返回值类型：Optional。这也是一个模仿 Scala 语言中的概念，作为一个容器，它可能含有某值，或者不包含。使用它的目的是尽可能避免 NullPointerException。

清单 14. Optional 的两个用例

```
String strA = " abcd ", strB = null;
print(strA);
print("");
print(strB);
getLength(strA);
getLength("");
getLength(strB);
public static void print(String text) {
 // Java 8
 Optional.ofNullable(text).ifPresent(System.out::println);
 // Pre-Java 8
 if (text != null) {
 System.out.println(text);
 }
 }
public static int getLength(String text) {
 // Java 8
return Optional.ofNullable(text).map(String::length).orElse(-1);
 // Pre-Java 8
// return if (text != null) ? text.length() : -1;
 };
```

在更复杂的 if (xx != null) 的情况中，使用 Optional 代码的可读性更好，而且它提供的是编译时检查，能极大的降低 NPE 这种 Runtime Exception 对程序的影响，或者迫使程序员更早的在编码阶段处理空值问题，而不是留到运行时再发现和调试。

Stream 中的 findAny、max/min、reduce 等方法等返回 Optional 值。还有例如 IntStream.average() 返回 OptionalDouble 等等。

reduce

这个方法的主要作用是把 Stream 元素组合起来。它提供一个起始值（种子），然后依照运算规则（BinaryOperator），和前面 Stream 的第一个、第二个、第 n 个元素组合。从这个意义上说，字符串拼接、数值的 sum、min、max、average 都是特殊的 reduce。例如 Stream 的 sum 就相当于

Integer sum = integers.reduce(0, (a, b) -> a+b); 或

Integer sum = integers.reduce(0, Integer::sum);

也有没有起始值的情况，这时会把 Stream 的前面两个元素组合起来，返回的是 Optional。

清单 15. reduce 的用例

```
// 字符串连接，concat = "ABCD"
String concat = Stream.of("A", "B", "C", "D").reduce("", String::concat);
// 求最小值，minValue = -3.0
double minValue = Stream.of(-1.5, 1.0, -3.0, -2.0).reduce(Double.MAX_VALUE, Double::min);
// 求和，sumValue = 10, 有起始值
int sumValue = Stream.of(1, 2, 3, 4).reduce(0, Integer::sum);
// 求和，sumValue = 10, 无起始值
sumValue = Stream.of(1, 2, 3, 4).reduce(Integer::sum).get();
// 过滤，字符串连接，concat = "ace"
concat = Stream.of("a", "B", "c", "D", "e", "F").
 filter(x -> x.compareTo("Z") > 0).
 reduce("", String::concat);
```

上面代码例如第一个示例的 reduce()，第一个参数（空白字符）即为起始值，第二个参数（String::concat）为 BinaryOperator。这类有起始值的 reduce() 都返回具体的对象。而对于第四个示例没有起始值的 reduce()，由于可能没有足够的元素，返回的是 Optional，请留意这个区别。

limit/skip

limit 返回 Stream 的前面 n 个元素；skip 则是扔掉前 n 个元素（它是由一个叫 subStream 的方法改名而来）。

清单 16. limit 和 skip 对运行次数的影响

```
public void testLimitAndSkip() {
 List<Person> persons = new ArrayList();
 for (int i = 1; i <= 10000; i++) {
 Person person = new Person(i, "name" + i);
 persons.add(person);
 }
List<String> personList2 = persons.stream().
map(Person::getName).limit(10).skip(3).collect(Collectors.toList());
 System.out.println(personList2);
}
private class Person {
 public int no;
 private String name;
 public Person (int no, String name) {
 this.no = no;
 this.name = name;
 }
 public String getName() {
 System.out.println(name);
 return name;
 }
}
```

输出结果为：

```
name1
name2
name3
name4
name5
name6
```

```
 7  name7
 8  name8
 9  name9
10  name10
11  [name4, name5, name6, name7, name8, name9, name10]
```

这是一个有 10，000 个元素的 Stream，但在 short-circuiting 操作 limit 和 skip 的作用下，管道中 map 操作指定的 getName() 方法的执行次数为 limit 所限定的 10 次，而最终返回结果在跳过前 3 个元素后只有后面 7 个返回。

有一种情况是 limit/skip 无法达到 short-circuiting 目的的，就是把它们放在 Stream 的排序操作后，原因跟 sorted 这个 intermediate 操作有关：此时系统并不知道 Stream 排序后的次序如何，所以 sorted 中的操作看上去就像完全没有被 limit 或者 skip 一样。

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

```
1  List<Person> persons = new ArrayList();
2   for (int i = 1; i <= 5; i++) {
3   Person person = new Person(i, "name" + i);
4   persons.add(person);
5   }
6  List<Person> personList2 = persons.stream().sorted((p1, p2) ->
7  p1.getName().compareTo(p2.getName())).limit(2).collect(Collectors.toList());
8  System.out.println(personList2);
```

上面的示例对清单 13 做了微调，首先对 5 个元素的 Stream 排序，然后进行 limit 操作。输出结果为：

```
1  name2
2  name1
3  name3
4  name2
5  name4
6  name3
7  name5
8  name4
9  [stream.StreamDW$Person@816f27d, stream.StreamDW$Person@87aac27]
```

即虽然最后的返回元素数量是 2，但整个管道中的 sorted 表达式执行次数没有像前面例子相应减少。

最后有一点需要注意的是，对一个 parallel 的 Steam 管道来说，如果其元素是有序的，那么 limit 操作的成本会比较大，因为它的返回对象必须是前 n 个也有一样次序的元素。取而代之的策略是取消元素间的次序，或者不要用 parallel Stream。

sorted

对 Stream 的排序通过 sorted 进行，它比数组的排序更强之处在于你可以首先对 Stream 进行各类 map、filter、limit、skip 甚至 distinct 来减少元素数量后，再排序，这能帮助程序明显缩短执行时间。我们对清单 14 进行优化：

清单 18. 优化：排序前进行 limit 和 skip

```
1  List<Person> persons = new ArrayList();
2   for (int i = 1; i <= 5; i++) {
3   Person person = new Person(i, "name" + i);
4   persons.add(person);
5   }
6  List<Person> personList2 = persons.stream().limit(2).sorted((p1, p2) -> p1.getName().compareTo(p2.getName()))
7  System.out.println(personList2);
```

结果会简单很多：

```
1  name2
2  name1
3  [stream.StreamDW$Person@6ce253f1, stream.StreamDW$Person@53d8d10a]
```

当然，这种优化是有 business logic 上的局限性的：即不要求排序后再取值。

min/max/distinct

min 和 max 的功能也可以通过对 Stream 元素先排序，再 findFirst 来实现，但前者的性能会更好，为 O(n)，而 sorted 的成本是 O(n log n)。同时它们作为特殊的 reduce 方法被独立出来也是因为求最大最小值是很常见的操作。

清单 19. 找出最长一行的长度

```
BufferedReader br = new BufferedReader(new FileReader("c:\\SUService.log"));
int longest = br.lines().
 mapToInt(String::length).
 max().
 getAsInt();
br.close();
System.out.println(longest);
```

下面的例子则使用 distinct 来找出不重复的单词。

清单 20. 找出全文的单词，转小写，并排序

```
List<String> words = br.lines().
 flatMap(line -> Stream.of(line.split(" "))).
 filter(word -> word.length() > 0).
 map(String::toLowerCase).
 distinct().
 sorted().
 collect(Collectors.toList());
br.close();
System.out.println(words);
```

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。

清单 21. 使用 Match

```
List<Person> persons = new ArrayList();
persons.add(new Person(1, "name" + 1, 10));
persons.add(new Person(2, "name" + 2, 21));
persons.add(new Person(3, "name" + 3, 34));
persons.add(new Person(4, "name" + 4, 6));
persons.add(new Person(5, "name" + 5, 55));
boolean isAllAdult = persons.stream().
 allMatch(p -> p.getAge() > 18);
System.out.println("All are adult? " + isAllAdult);
boolean isThereAnyChild = persons.stream().
 anyMatch(p -> p.getAge() < 12);
System.out.println("Any child? " + isThereAnyChild);
```

输出结果：

```
All are adult? false
Any child? true
```

进阶：自己生成流

Stream.generate

通过实现 Supplier 接口，你可以自己来控制流的生成。这种情形通常用于随机数、常量的 Stream，或者需要前后元素间维持着某种状态信息的 Stream。把 Supplier 实例传递给 Stream.generate() 生成的 Stream，默认是串行（相对 parallel 而言）但无序的（相对 ordered 而言）。由于它是无限的，在管道中，必须利用 limit 之类的操作限制 Stream 大小。

清单 22. 生成 10 个随机整数

```
Random seed = new Random();
Supplier<Integer> random = seed::nextInt;
Stream.generate(random).limit(10).forEach(System.out::println);
//Another way
IntStream.generate(() -> (int) (System.nanoTime() % 100)).
limit(10).forEach(System.out::println);
```

Stream.generate() 还接受自己实现的 Supplier。例如在构造海量测试数据的时候，用某种自动的规则给每一个变量赋值；或者依据公式计算 Stream 的每个元素值。这些都是维持状态信息的情形。

清单 23. 自实现 Supplier

```
Stream.generate(new PersonSupplier()).
limit(10).
forEach(p -> System.out.println(p.getName() + ", " + p.getAge()));
private class PersonSupplier implements Supplier<Person> {
 private int index = 0;
 private Random random = new Random();
 @Override
 public Person get() {
 return new Person(index++, "StormTestUser" + index, random.nextInt(100));
 }
}
```

输出结果：

```
StormTestUser1, 9
StormTestUser2, 12
StormTestUser3, 88
StormTestUser4, 51
StormTestUser5, 22
StormTestUser6, 28
StormTestUser7, 81
StormTestUser8, 51
StormTestUser9, 4
StormTestUser10, 76
```

Stream.iterate

iterate 跟 reduce 操作很像，接受一个种子值，和一个 UnaryOperator（例如 f）。然后种子值成为 Stream 的第一个元素，f(seed) 为第二个，f(f(seed)) 第三个，以此类推。

清单 24. 生成一个等差数列

```
Stream.iterate(0, n -> n + 3).limit(10). forEach(x -> System.out.print(x + " "));.
```

输出结果：

```
0 3 6 9 12 15 18 21 24 27
```

与 Stream.generate 相仿，在 iterate 时候管道必须有 limit 这样的操作来限制 Stream 大小。

进阶：用 Collectors 来进行 reduction 操作

java.util.stream.Collectors 类的主要作用就是辅助进行各类有用的 reduction 操作，例如转变输出为 Collection，把 Stream 元素进行归组。

groupingBy/partitioningBy

清单 25. 按照年龄归组

```
Map<Integer, List<Person>> personGroups = Stream.generate(new PersonSupplier()).
 limit(100).
 collect(Collectors.groupingBy(Person::getAge));
Iterator it = personGroups.entrySet().iterator();
while (it.hasNext()) {
```

```
 Map.Entry<Integer, List<Person>> persons = (Map.Entry) it.next();
 System.out.println("Age " + persons.getKey() + " = " + persons.getValue().size());
}
```

上面的 code，首先生成 100 人的信息，然后按照年龄归组，相同年龄的人放到同一个 list 中，可以看到如下的输出：

```
Age 0 = 2
Age 1 = 2
Age 5 = 2
Age 8 = 1
Age 9 = 1
Age 11 = 2
……
```

清单 26. 按照未成年人和成年人归组

```
Map<Boolean, List<Person>> children = Stream.generate(new PersonSupplier()).
 limit(100).
 collect(Collectors.partitioningBy(p -> p.getAge() < 18));
System.out.println("Children number: " + children.get(true).size());
System.out.println("Adult number: " + children.get(false).size());
```

输出结果：

```
Children number: 23
Adult number: 77
```

在使用条件“年龄小于 18”进行分组后可以看到，不到 18 岁的未成年人是一组，成年人是另外一组。partitioningBy 其实是一种特殊的 groupingBy，它依照条件测试的是否两种结果来构造返回的数据结构，get(true) 和 get(false) 能即为全部的元素对象。

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。

相关主题

- Oracle Java 8 官方文档对 java.util.stream package 的说明。
- 一篇教程：Java 8 Tutorials, Resources, Books and Examples to learn Lambdas, Stream API and Functional Interfaces。
- 关于这篇 Lambda 和 Stream 更多介绍的教程。
- 访问 IBM Developer (原 developerWorks) 中国 Java 专区，了解关于信息管理的更多信息，获取技术文档、how-to 文章、培训、下载、产品信息以及其他资源。