链滴

JDk 8 之 Stream 流 （一）

作者：alex18595752445

原文链接：https://ld246.com/article/1584380581548

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

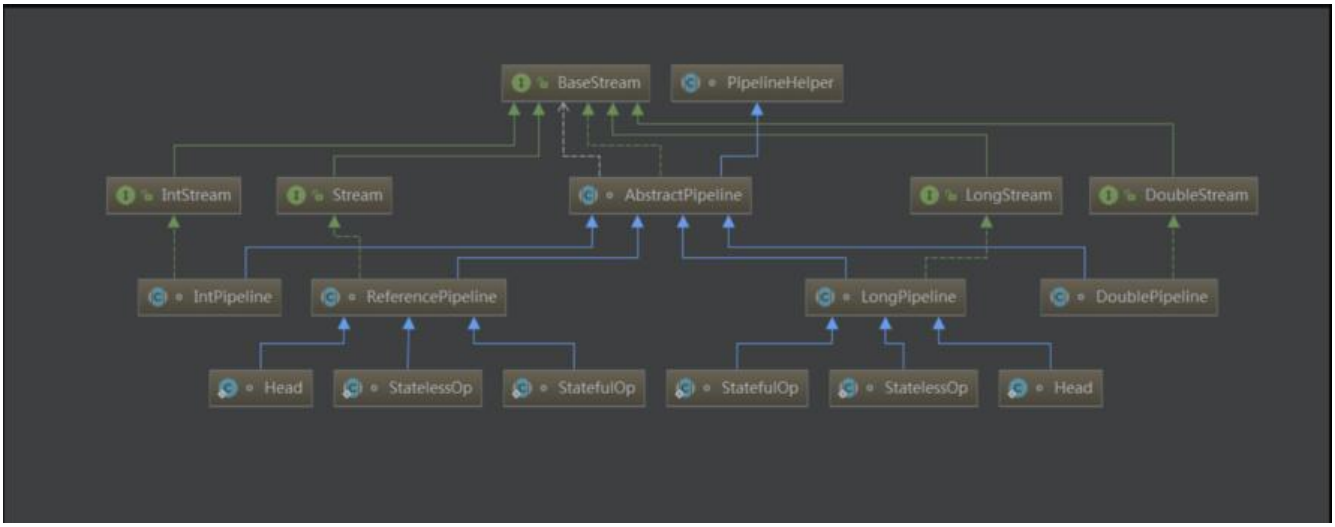
一、前言

在目前用到的 JDK8 的功能当中，毫无疑问 Stream 的使用是最多的，所以通过这篇文章来学习总一下。

首先，Java8 的 Stream 是对集合对象操作的 API，它专注于对集合对象进行各种非常便利，高效聚合操作或者大批量操作，从而减少代码的复杂度。借助于 lambda 表达式，极大的提高编程效率和序可读性。并且 Stream 支持串行和并行两种模式，使我们无需编写太多代码，就可以很方便的写出性能的并发程序。

先上一段前言，总结就一句话：使用普通的集合处理方式太费劲（只是为了一个简单排序，就要进行合遍历），所以 JDK 8 为了解决这个问题 引入了 stream ，就是对集合处理进行了一次封装。

二、Stream 结构及构建



```
public interface Stream<T> extends BaseStream<T, Stream<T>>
public interface BaseStream<T, S extends BaseStream<T, S>>
    extends AutoCloseable {
```

可以看到，Stream 继承自 BaseStream 接口，而 BaseStream 又继承自 AutoCloseable 接口，顾思义，AutoCloseable 负责流的自动关闭。

我们这里来了解下生成 Stream 的几种常用方式：

```
// 1. 借助Stream的of方法
Stream stream = Stream.of("a", "b", "c");
String [] strArray = new String[] {"a", "b", "c"};
// 2. 通过数组生成Stream
stream = Stream.of(strArray);
stream = Arrays.stream(strArray);
// 3. 通过集合来生成Stream
List<String> list = Arrays.asList(strArray);
stream = list.stream();
```

而对于基础数值类型，目前提供了三种对应的包装类型 Stream：IntStream，LongStream，Double trem，当然我们也可以使用 Stream<Integer>、Stream<Long>、Stream<Double>，但是 boxin 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。

三、Stream 使用

先说下 Stream 的类型，Stream 一般情况下包含了两个类型：中间操作(Intermediate)和结束操作(Teminal)：

- **Intermediate**，所谓的中间操作，就是说每次调用做一些处理之后会返回一个新的 Stream，这类作都是惰性的，也就是说并没有真正开始流的遍历。这些操作包括：map (mapToInt, flatMap 等)、 filter、 distinct、 sorted、 peek、 limit、 skip、 parallel 等；
- **Terminal**，一个 Stream 只能执行一次结束操作，而且只能是最后一个操作，执行 terminal 之后 Stream 被消费掉了，并且产生了一个结果，这些操作包括：forEach、 forEachOrdered、 toArray、 reduce、 collect、 min、 max、 count、 anyMatch、 allMatch、 noneMatch、 findFirst、 findny 等；

再简单说下 Stream 流的特点，Stream 其实有点类似于迭代器，每个 Stream 只能操作一次，操作之后就不能再操作该对象了，也就是一种单向的，不可重复操作的对象。

2. map 方法

```
List<String> list = Arrays.asList("stream", "map");
Stream<String> stream = list.stream();
List<String> newList = stream.map(input -> input.toUpperCase()).collect(Collectors.toList());
```

因为 Stream 只能使用一次，如果我们再操作的话就是抛出异常（这就是 流只能被消费一次的原因。tream 就类比成一个杯子，杯子里的水就像 Stream 里的数据，你把杯子里的水拿出来了，杯子的水没有了，Stream 也是同样的道理）：

```
List<String> newList = stream.map(input -> input.toUpperCase()).collect(Collectors.toList());
newList = stream.map(input -> input.toLowerCase()).collect(Collectors.toList());

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated
```

```
pon or closed
    at java.util.stream.AbstractPipeline.<init>(AbstractPipeline.java:203)
    at java.util.stream.ReferencePipeline.<init>(ReferencePipeline.java:94)
    at java.util.stream.ReferencePipeline$StatelessOp.<init>(ReferencePipeline.java:618)
    at java.util.stream.ReferencePipeline$3.<init>(ReferencePipeline.java:187)
    at java.util.stream.ReferencePipeline.map(ReferencePipeline.java:186)
```

3. mapToInt/mapToLong/mapToDouble 方法

顾名思义，这就是将对应的 Stream 转为 IntStream，LongStream，DoubleStream

```
List<Integer> list = Arrays.asList(100, 200);
IntStream intStream = list.stream().mapToInt(input -> input);
```

4. flatMap 方法

前面说过的 map 方法是一对一的输入输出，而 flatMap 方法则是一种一对多的映射关系。

```
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());
List<Integer> list = outputStream.collect(Collectors.toList());
System.out.println(list);
```

```
[1,2,3,4,5,6]
```

flatMap 是对 input Stream 中的层级进行结构扁平化，就是将最底层元素抽出来放到一起，最终 out ut 的新 Stream 里面都是单个的数字。我们再来简单看下 map 方法的对应实现：

```
// inputStream不变
Stream<Stream<Integer>> stream = inputStream.map(childList -> childList.stream());
List<List<Integer>> list = stream.map(input -> input.collect(Collectors.toList())).collect(Collec
ors.toList());
System.out.println(list);
```

```
[[1], [2, 3], [4, 5, 6]]
```

从这里可以大致看出它们的区别，对于 flatMap 来说，它的输入输出大致如下：

```
{{1,2}，{3,4}，{5,6}} -> flatMap  -> {1,2,3,4,5,6}
```

而对 map 方法来说，则是：

```
{{1,2}，{3,4}，{5,6}} -> map -> {1,2}, {3,4},{5,6}
```

5. filter 方法

该方法用于对 Stream 中的元素按照某些条件进行过滤，过滤后的元素生成一个新的元素，比如过滤组中的偶数：

```
Integer[] sixNums = {1, 2, 3, 4, 5, 6};
Stream.of(sixNums).filter(n -> (n % 2 == 0)).forEach(num -> System.out.print(num + " "));
```

```
2 4 6
```

6. foreach 方法

类似于 for 循环，用于遍历 Stream 中的每个元素，比较简单，可能需要注意的是，forEach 不能修自己包含的本地变量值，也不能使用 break/return 之类的关键字提前结束循环：

```
Stream<String> stream = Stream.of("hello", "world");
// 方式1
stream.forEach(num -> System.out.print(num));
// 方式2
stream.forEach(System.out::print);
```

7. findFirst

返回 Stream 对象的第一个元素，由于返回的是 Optional，所以返回的值有可能为空：

```
Stream<String> stream = Stream.of("hello", "world");
Optional<String> optional = stream.findFirst();
String name = optional.map(String::toLowerCase).orElse("");
System.out.println(name);
```

Optional 是 jdk8 提供的一种用于优雅的解决 NullPointExecption 的方式，等下篇文章我们来学习下。

8. reduce 方法

这个方法的作用主要是把 Stream 中的元素组合起来，比如说字符串拼接，数值类型的求和等都是特的 reduce 操作，并且我们可以根据重载方法选择是否有初始值。

```
// 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);
// 求和的另一种形式
int sum = Stream.of(1, 2, 3, 4).reduce(0, (a,b) -> a+b);
// 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值的 reduce()，返回的是 Optional，请留意这个区别。

9. limit/skip 方法

limit 方法用于返回 Stream 元素的前 n 个元素，而 skip 方法是跳过前 n 个元素返回剩余的元素：

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
list.stream().limit(8).forEach(System.out::print);
System.out.println();
list.stream().limit(8).skip(3).forEach(System.out::print);
```

```
12345678
45678
```

10. sorted

sorted 方法是用于对 Stream 元素进行排序的，我们可以按照默认的自然排序规则进行排序， 也可指定具体的比较器来进行排序：

```
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
```

Stream 的 sorted 方法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 操作之后再进行排序：

```
List<Integer> list = Arrays.asList(5, 7, 1, 4, 2, 6, 3, 8, 9, 10);
list.stream().limit(5).sorted().forEach(System.out::print);
System.out.println();
list.stream().limit(5).sorted(Comparator.reverseOrder()).forEach(System.out::print);
```

```
12457
75421
```

11. min/max/distinct 方法

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，O(n)，而 sorted 的成本是 O(n log n)。我们来看一下获取最小值的方式：

```
List<Integer> list = Arrays.asList(5, 7, 1, 4, 2, 6, 3, 8, 9, 10);
// 通过Stream的min方法
Integer min2 = list.stream().min(Comparator.naturalOrder()).get();
// 通过IntStrem的min方法
Integer min1 = list.stream().mapToInt(input -> input).min().getAsInt();
```

而 distinct 方法是用于过滤重复数据的：

```
List<Integer> list = Arrays.asList(5, 7, 1, 7, 2, 6, 3, 9, 9, 10);
list.stream().distinct().forEach(System.out::print);
```

```
571263910
```

12. allMatch/anyMatch/noneMatch 方法

● allMatch， 对 Stream 中的所有元素进行判断，全部满足条件的时候返回 true，只要有一个不符条件就返回 false；

- anyMatch，Stream 中只要有一个满足条件，就返回 true；
- noneMatch，Stream 中没有一个元素满足条件，这时返回 true；

```
List<Integer> list = Arrays.asList(5, 7, 1, 7, 2, 6, 3, 9, 9, 10);
boolean isAllRight = list.stream().allMatch(input -> (input > 0));
boolean isAnyRight = list.stream().anyMatch(input -> (input > 2));
boolean isNoneRight = list.stream().noneMatch(input -> (input < 0));
System.out.println("isAllRight:" + isAllRight + " isAnyRight:" + isAnyRight + " isNoneRight:" +
sNoneRight);
```

```
isAllRight:true isAnyRight:true isNoneRight:true
```

13. peek 方法

peek 方法，会生成一个包含原 Stream 的所有元素的新 Stream，同时会提供一个消费函数（Consumer 实例），新 Stream 每个元素被消费的时候都会执行给定的消费函数。比如说 foreach 方法是一 terminal 操作，执行之后，Stream 就消费掉了，我们无法对一个 Stream 进行两次操作，而 peek 法作为 intermediate 操作，则可以达到类似的目的：

```
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());
```

```
Filtered value: three
Mapped value: THREE
Filtered value: four
Mapped value: FOUR
```

如上，我们可以在遍历列表的时候，先打印字符串，再将该字符串转成大写再打印出来。

另外，根据 API 的说明，该方法主要用于调试，方便 debug 查看 Stream 内进行处理的每个元素。

14. forEachOrdered 方法

forEachOrdered 方法和 forEach 方法功能一样，都是用于遍历 Stream，不同的地方在于并行流处理上。并行的时候 forEach 方法为了效率，它的顺序和 Stream 元素的顺序不一定完全一样，而 forachOrdered 方法的顺序则是和 Stream 元素的顺序是一样的

```
List<String> list = Arrays.asList("x", "y", "z");

list.parallelStream().forEach(x -> System.out.print(" " + x));
System.out.println();
list.parallelStream().forEachOrdered(x -> System.out.print(" " + x));
System.out.println();

//输出的顺序不一定（效率更高）
Stream.of("AAA", "BBB", "CCC").parallel().forEach(s -> System.out.print(" " + s));
System.out.println();
//输出的顺序与元素的顺序严格一致
Stream.of("AAA", "BBB", "CCC").parallel().forEachOrdered(s -> System.out.print(" " + s));
```

```
y x z
x y z
BBB CCC AAA
AAA BBB CCC
```

15. toArray 方法

这个方法比较简单，就是返回对应的数组，该方法默认是返回 Object 数组，不过我们可以使用它的载方法返回对应格式的数组

```
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> generator);
```

Eg：

```
List<String> list = Arrays.asList("x", "y", "z");
Object[] objects = list.stream().toArray();
Integer[] arrays = list.stream().toArray(Integer[]::new);
```

16. count 方法

count 方法表示获取 Stream 流中元素的数量，返回 long 类型：

```
// 打印 4
long num = Stream.of(1, 2, 3, 4).count();
// 打印 3
long num = Stream.of(1, 2, 3, 4).limit(3).count();
```

17. findAny 方法

findAny 方法表示从流中随便选择一个元素，该方法返回的值是不稳定的：

```
Integer num = Stream.of(1, 2, 3, 4).findAny().get();
```

18. collect 方法

collect 方法我们前面已经接触过，有两个方法，我们先看一下简单的那个：

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

在前文中，我们使用 map 方法对流进行处理之后，返回的还是一个 Stream，而此时我们是无法我的集合操作的，这时候就需要将流重新转换为集合框架中对应的集合，那么这时候我们就可以通过该法来实现：

```
List<String> list = Arrays.asList("hello", "world").stream().collect(Collectors.toList());
```

该方法接收一个 Collector 类型的参数，但幸运的是 Java8 给我们提供了 Collector 的工具类：Collectors，这其中已经定义了一些静态工厂方法，比如 Collectors.toCollection() 生成集合，Collectors.toList() 生成 List，Collectors.toSet() 生成 Set 等，Collectors 是个很好的工具类，封装了许多操作，后我们再来介绍。

接下来，再简单看下该方法的另一个重载方法：

```
<R> R collect(Supplier<R> supplier,
    BiConsumer<R, ? super T> accumulator,
    BiConsumer<R, R> combiner);
```

该方法比较复杂，我们先简单分析下，等以后如果用到了，再来仔细研究。该方法有三个参数：Supplier supplier 是一个工厂函数，用来生成一个新的容器；BiConsumer accumulator 也是一个函数，来把 Stream 中的元素添加到结果容器中；BiConsumer combiner 还是一个函数，用来把中间状态多个结果容器合并成为一个（并发的时候会用到）。来简单看一下例子吧：

```
List<Integer> nums = Arrays.asList(1, 1, null, 2, 3, 4, null, 5, 6, 7, 8, 9, 10);
List<Integer> numsWithoutNull = nums.stream().filter(num -> num != null).
      collect(() -> new ArrayList<Integer>(),
            (list, item) -> list.add(item),
            (list1, list2) -> list1.addAll(list2));
```

使用方法引用来优化下该例子：

```
List<Integer> nums = Arrays.asList(1, 1, null, 2, 3, 4, null, 5, 6, 7, 8, 9, 10);
List<Integer> numsWithoutNull = nums.stream().filter(num -> num != null).
      collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

接下来，说下该方法：该方法是将一个 Integer 类型的 List，先过滤掉为 null 的元素，然后把剩下的素放到新的 List 中。再来看一下这些参数：

- 1. 第一个函数生成一个新的 ArrayList 实例；
- 1. 第二个函数接受两个参数，第一个是前面生成的 ArrayList 对象，第二个是 stream 中包含的元，函数体就是把 stream 中的元素加入 ArrayList 对象中。第二个函数被反复调用直到原 stream 的素被消费完毕；
- 1. 第三个函数也是接受两个参数，这两个都是 ArrayList 类型的，函数体就是把第二个 ArrayList 部加入到第一个中；

这么来看，这个方法是有点复杂，并且单看这个例子的话，是完全可以使用上面那个重载方法然后借助 Collectors.toList 来实现的。对这个方法的了解就到这了，等以后如果用到了，再来更新。

接下来是 Stream 的静态方法，这些静态方法目的都是为了创建 Stream 流。

1. of 方法

Stream 的 of 方法用来构建有序的 Stream 对象，有两个方法，提供单个对象及多个对象的构建：

```
public static<T> Stream<T> of(T t)
public static<T> Stream<T> of(T... values)
```

比如说：

```
Stream stream = Stream.of("a");
IntStream intStream= IntStream.of(1, 2, 3);
```

2. builder 方法

通过使用 Stream.builder 方法生成 Builder 对象，Builder 对象是 Stream 的可变构造器，也称为构造器，该对象允许单独生成元素并添加到构造器，然后来生成流，来避免使用 ArrayList 作为临时冲区产生的复制开销。

流构建器有一个生命周期，它从一个构建阶段开始，在这个阶段中可以添加元素，然后过渡到一个建阶段，在这个阶段之后，可能不会添加元素。构建阶段从调用 build()方法开始，该方法创建一个序流，其元素是按照添加到流构建器的顺序添加到流构建器的元素。

```
Stream.Builder builder = Stream.builder();
builder.accept("hello");
builder.add("world");
Stream stream = builder.build();
stream.forEach(input -> System.out.print(input + " "));

//或者
Stream<String> streamBuilder = Stream.<String>builder().add("hello").add("world").build();
```

```
hello world
```

3. empty 方法

创建一个不包含任何元素的有序的 Stream 流：

```
Stream<Integer> stream = Stream.empty();
```

4. iterate 方法

Stream 的 iterate 方法和 reduce 方法有点像，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：

```
// 比如生成等差数列 0 3 6 9 12 15 18 21 24 27
Stream.iterate(0, n -> n + 3).limit(10). forEach(x -> System.out.print(x + " "));
```

同样，iterate 也是无限的，在进行 iterate 的时候，必须要有 limit 这样的操作来限制大小，但 iterat 生成的 Stream 是连续且有序的。

5. generate 方法

通过实现 Supplier 接口，我们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 Strea ，把 Supplier 实例传递给 Stream.generate() ，这种生成的 Stream 流是无限的，所以我们必须使用 imit 等方法来限制 Stream 的大小，并且通过 generate 方法生成的 Stream 是无序的；

```
// 生成10个随机数
Stream.generate(new Random()::nextInt).limit(10).forEach(System.out::println);
//另外一种方式
IntStream.generate(() -> (int) (System.nanoTime() % 100)).limit(10).forEach(System.out::println)
```

6. concat 方法

返回两个 Stream 流连接的流：

```
public static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)

Stream<Integer> stream1 = Stream.of(1, 2, 3, 4);
Stream<Integer> stream2 = Stream.of(5, 6, 7, 8);
Stream.concat(stream1, stream2).forEach(System.out::print);
```

四、IntStream LongStream DoubleStream 补充

因为这三个 Stream 中的操作属于数值操作，所以它们中有些方法 Stream 中并没有，我们也来简单绍下。由于这三个 Stream 都差不多，我们就以 IntStream 来进行举例。

1. sum/min/max/count/average 方法

min，max，count 方法 Stream 中都有，只不过在 IntStream 中这些方法的参数和返回值可能和 Stream 方法的返回值有稍许的不同，不过功能是一样的。

```
// 计算min
IntStream.of(1, 2, 3, 4).min().getAsInt();
IntStream.of(1, 2, 3, 4).reduce(0, Integer::min);

// 计算max
IntStream.of(1, 2, 3, 4).max().getAsInt();
IntStream.of(1, 2, 3, 4).reduce(0, Integer::max);
```

sum 方法的话，就是用于计算 Stream 中元素值的和，同样也可也使用 reduce 方法来代替：

```
int sum = IntStream.of(1, 2, 3, 4).sum();
int sum = IntStream.of(1, 2, 3, 4).reduce(0, Integer::sum);
```

而 average 方法是用于计算平均值的，返回的是 OptionalDouble 类型：

```
// 计算平均值
double average = IntStream.of(1, 2, 3, 4).average().getAsDouble();
```

2. summaryStatistics 方法

该方法用于获取 Stream 流的各项汇总数据，我们直接看例子就明白了：

```
// 各种计算值的汇总数据
IntSummaryStatistics summaryStatistics = IntStream.of(1, 2, 3, 4).summaryStatistics();
// 平均值，元素个数，最大值，最小值，总和
System.out.println(summaryStatistics.getAverage());
System.out.println(summaryStatistics.getCount());
System.out.println(summaryStatistics.getMax());
System.out.println(summaryStatistics.getMin());
System.out.println(summaryStatistics.getSum());
```

3. asLongStream/asDoubleStream 方法

这两个方法比较简单，就是转为对应的 LongStream 流和 DoubleStream 流。

4. boxed 方法

基础类型的装箱操作，比如将 int 类型装箱称为 Integer 类型：

```
Stream<Integer> stream = IntStream.of(1, 2, 3, 4).boxed();
```

5. range 方法

range 方法是 IntStream 中的静态方法，用于构建某段范围的 IntStream 流：

```
public static IntStream range(int startInclusive, int endExclusive)
```

创建的 Stream 流包含开始值 startInclusive(**inclusive**)，但不包含结束值 endExclusive(**exclusive**)：

```
IntStream intStream = IntStream.range(1, 5);
// 1 2 3 4
intStream.forEach(x -> System.out.print(x + " "));
```

6. rangeClosed 方法

rangeClosed 方法和 range 方法唯一的不同就是，创建的 Stream 流既包含开始值，又包含结束值这点从参数命名上就可以知道。不得不说，该方法参数的命名很规范，值得我们学习：

```
public static IntStream rangeClosed(int startInclusive, int endInclusive)
```

对应的实例：

```
IntStream intStream = IntStream.rangeClosed(1, 5);
// 1 2 3 4 5
intStream.forEach(x -> System.out.print(x + " "));
```

五、BaseStream 中的方法

上面忘了说了，BaseStream 作为 Stream 的底层接口，有几个方法值得了解一下：

1. parallel 方法

返回一个并行的且等效流，可能返回该流本身，因为该 Stream 已经是并行的，或者该 Stream 的底状态被修改为了并行。

2. isParallel 方法

判断该 Stream 是否是并行的：

```
IntStream intStream = IntStream.rangeClosed(1, 5);
// false
boolean isParallel = intStream.isParallel();
// true
isParallel = intStream.parallel().isParallel();
```

3. iterator/spliterator 方法

这两个方法就比较简单了，iterator 就是返回迭代器对象，而 spliterator 则是返回一个并行的迭代器象；

4. unordered 方法

返回一个无序的等效的 Stream，可能返回的是 Stream 本身，因为该 Stream 已经是无序的，或者 Stream 的底层状态被修改为了无序。当不考虑流的顺序时，可以使用无序的 Stream 来进行操作，样可以加快一些方法的执行速度，提高一些性能，一般用于并行的时候。对于 unordered 方法有个问题可参考：https://stackoverflow.com/questions/21350195/stream-ordered-unordered-problems/38038578

https://coderanch.com/t/692966/certification/unordered-streams