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。

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操作如：取平均值、取最大最小值、取汇总值、或者进行分组等等类似的这些操作。

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，如果要找一年级的所有学生，然后返回按学生分数值降序排序好的学生ID的集合，我们需要这样写：

```
public class Student {

    private Integer id;         // ID
    private Grade grade;        // 年纪
    private Integer score;      // 分数

    public Student(Integer id, Grade grade, Integer score) {
        this.id = id;
        this.grade = grade;
        this.score = score;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Grade getGrade() {
```

```
        return grade;
    }

    public void setGrade(Grade grade) {
        this.grade = grade;
    }

    public Integer getScore() {
        return score;
    }

    public void setScore(Integer score) {
        this.score = score;
    }
}
```

```
public enum  Grade {
    FIRST, SECOND, THTREE
}
```

```
    public static void main(String[] args) {
        final Collection<Student> students = Arrays.asList(
                new Student(1, Grade.FIRST, 60),
                new Student(2, Grade.SECOND, 80),
                new Student(3, Grade.FIRST, 100)
        );

        List<Student> gradeOneStudents = Lists.newArrayList();
        for (Student student: students) {
            if (Grade.FIRST.equals(student.getGrade())) {
                gradeOneStudents.add(student);
            }
        }

        Collections.sort(gradeOneStudents, new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
                return o2.getScore().compareTo(o1.getScore());
            }
        });

        List<Integer> studentIds = new ArrayList<>();
        for(Student t: gradeOneStudents){
            studentIds.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。

```
public static void main(String[] args) {
    final Collection< Student > students= Arrays.asList(
            new Student(1, Grade.FIRST, 60),
            new Student(2, Grade.SECOND, 80),
            new Student(3, Grade.FIRST, 100)
    );

    List<Integer> studentIds = students.stream()
            .filter(student -> student.getGrade().equals(Grade.FIRST))
            .sorted(Comparator.comparingInt(Student::getScore))
            .map(Student::getId)
            .collect(Collectors.toList());
}
```

什么是Stream

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。

Stream的特点

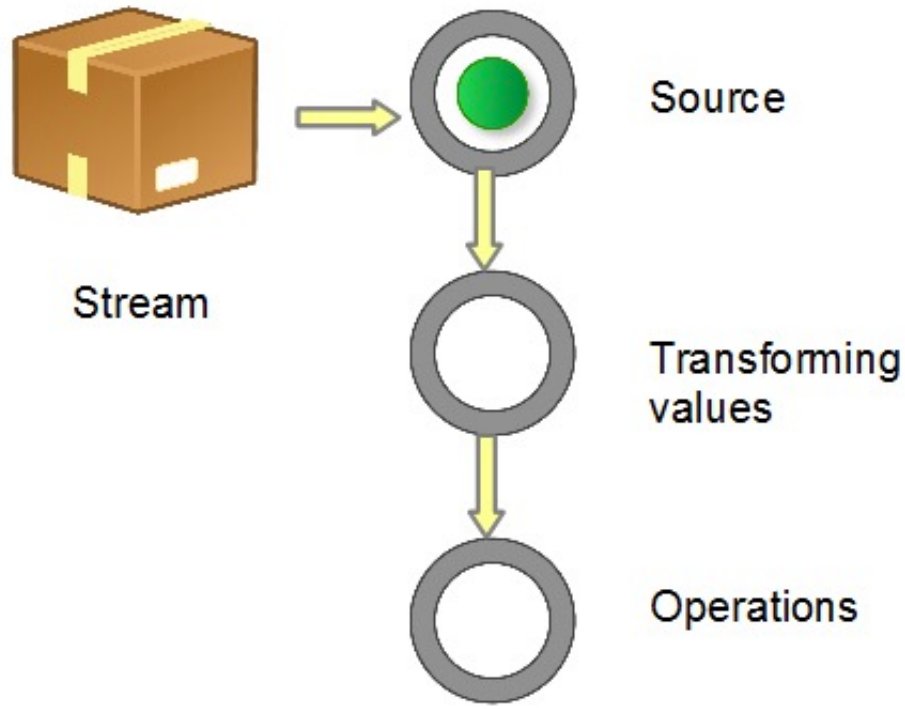
1. Iterator，用户只能显式地一个一个遍历元素并对其执行某些操作；Stream，用户只要给出需要对其包含的元素执行什么操作，比如 “过滤掉长度大于 10 的字符串”、“获取每个字符串的首字母”等，Stream 会隐式地在内部进行遍历，做出相应的数据转换。
2. Stream 就如同一个Iterator，单向，不可往复，数据只能遍历一次，遍历过一次后即用尽了，就好比流水从面前流过，一去不复返。
3. Stream 可以并行化操作，Iterator只能命令式地、串行化操作。顾名思义，当使用串行方式去遍历时，每个 item 读完后再读下一个 item。而使用并行去遍历时，数据会被分成多个段，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 框架来拆分任务和加速处理过程。

Stream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



生成Stream Source的方式：

- 从Collection 和数组生成
  - Collection.stream()
  - Collection.parallelStream()
  - Arrays.stream(T array)
  - Stream.of(T t)
- 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()

Stream的操作类型

- 中间操作(Intermediate Operation)：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。
- 终止操作(Terminal Operation)：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。

Intermediate Operation又可以分为两种类型：

- 无状态操作(Stateless Operation)：操作是无状态的，不需要知道集合中其他元素的状态，每个元素之间是相互独立的，比如map()、filter()等操作。
- 有状态操作(Stateful Operation)：有状态操作，操作是需要知道集合中其他元素的状态才能进行的，比如sort()、distinct()。

Terminal Operation从逻辑上可以分为两种：

- 短路操作(short-circuiting)：短路操作是指不需要处理完所有元素即可结束整个过程
- 非短路操作(non-short-circuiting)：非短路操作是需要处理完所有元素之后才能结束整个过程

Stream的使用

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。

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、Stream >、Stream，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。

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);
```

流转换为其它数据结构

```
Stream<String> stream = Stream.<String>of(new String[]{"1", "2", "3"});
List<String> list1 = stream.collect(Collectors.toList());
List<String> list2 = stream.collect(Collectors.toCollection(ArrayList::new));

String str = stream.collect(Collectors.joining());
System.out.println(str);
```

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。

例子：

```
        Stream<String> stream = Stream.<String>of(new String[]{"a", "b", "c"});
        stream.map(String::toUpperCase).forEach(System.out::println);
```

这段代码把所有的字母转换为大写。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。

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

        Stream<Integer> mapStream = inputStream.map(List::size);
        Stream<Integer> flatMapStream = inputStream.flatMap(Collection::stream);
```

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。

```
        Integer[] nums = new Integer[]{1,2,3,4,5,6};
        Arrays.stream(nums).filter(n -> n<3).forEach(System.out::println);
```

将小于3的数字留下来。

forEach

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。下面的代码会报错。

```
Integer[] nums = new Integer[]{1,2,3,4,5,6};
Stream stream = Arrays.stream(nums);
stream.forEach(System.out::print);
stream.forEach(System.out::print);
```

相反，具有相似功能的 intermediate 操作 peek 可以达到上述目的。

```
        Integer[] nums = new Integer[]{1,2,3,4,5,6};
        Stream stream = Arrays.stream(nums);
        stream
                .peek(System.out::print)
                .peek(System.out::print)
                .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。下面的代码还是打印出所有元素，并不会提前返回。

```
        Integer[] nums = new Integer[]{1,2,3,4,5,6};
        Arrays.stream(nums).forEach(integer -> {
            System.out.print(integer);
            return;
        });
```

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。

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[] nums = new Integer[]{1,2,3,4,5,6};

        Integer sum = Arrays.stream(nums).reduce(0, (integer, integer2) ->
integer+integer2);
        System.out.println(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。

```
        Integer[] nums = new Integer[]{1,2,3,4,5,6};
        // 有初始化值
        Integer sum = Arrays.stream(nums).reduce(0, Integer::sum);
        // 无初始化值
        Integer sum1 = Arrays.stream(nums).reduce(Integer::sum).get();
```

limit/skip

limit 返回 Stream 的前面 n 个元素；skip 则是扔掉前 n 个元素。

```
        Integer[] nums = new Integer[]{1,2,3,4,5,6};

        Arrays.stream(nums).limit(3).forEach(System.out::print);
        System.out.println();
        Arrays.stream(nums).skip(2).forEach(System.out::print);
```

结果：

```
123
3456
```

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。

```
        Integer[] nums = new Integer[]{1,2,3,4,5,6};


        Arrays.stream(nums).sorted((i1, i2) ->
i2.compareTo(i1)).limit(3).forEach(System.out::print);
        System.out.println();
        Arrays.stream(nums).sorted((i1, i2) ->
i2.compareTo(i1)).skip(2).forEach(System.out::print);
```

min/max/distinct

```
        Integer[] nums = new Integer[]{1, 2, 2, 3, 4, 5, 5, 6};


        System.out.println(Arrays.stream(nums).min(Comparator.naturalOrder()).get());
        System.out.println(Arrays.stream(nums).max(Comparator.naturalOrder()).get());
        Arrays.stream(nums).distinct().forEach(System.out::print);
```

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。

```
Integer[] nums = new Integer[]{1, 2, 2, 3, 4, 5, 5, 6};
System.out.println(Arrays.stream(nums).allMatch(integer -> integer < 7));
System.out.println(Arrays.stream(nums).anyMatch(integer -> integer < 2));
System.out.println(Arrays.stream(nums).noneMatch(integer -> integer < 0));
```

结果都为true。

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

例如对上面的Student进行按年级进行分组：

```
        final Collection<Student> students = Arrays.asList(
                new Student(1, Grade.FIRST, 60),
                new Student(2, Grade.SECOND, 80),
                new Student(3, Grade.FIRST, 100)
        );

        // 按年级进行分组

students.stream().collect(Collectors.groupingBy(Student::getGrade)).forEach(((grade,
students1) -> {
            System.out.println(grade);
            students1.forEach(student ->
System.out.println(student.getId()+","+student.getGrade()+","+student.getScore()));
        }));
```

结果：

```
FIRST
1,FIRST,60
3,FIRST,100
SECOND
2,SECOND,80
```

例如对上面的Student进行按分数段进行分组：

```
students.stream().collect(Collectors.partitioningBy(student -> student.getScore()
<=60)).forEach(((grade, students1) -> {
            System.out.println(grade);
            students1.forEach(student ->
System.out.println(student.getId()+","+student.getGrade()+","+student.getScore()));
        }));
```

结果：

```
false
2,SECOND,80
3,FIRST,100
true
1,FIRST,60
```

parallelStream

parallelStream其实就是一个并行执行的流.它通过默认的ForkJoinPool,可以提高你的多线程任务的速度。

parallelStream使用

```
Arrays.stream(nums).parallel().forEach(System.out::print);
System.out.println(Arrays.stream(nums).parallel().reduce(Integer::sum).get());
System.out.println();
Arrays.stream(nums).forEach(System.out::print);
System.out.println(Arrays.stream(nums).reduce(Integer::sum).get());
```

parallelStream要注意的问题

parallelStream底层是使用的ForkJoin。而ForkJoin里面的线程是通过ForkJoinPool来运行的，Java 8为ForkJoinPool添加了一个通用线程池，这个线程池用来处理那些没有被显式提交到任何线程池的任务。它是ForkJoinPool类型上的一个静态元素。它拥有的默认线程数量等于运行计算机上的处理器数量，所以这里就出现了这个java进程里所有使用parallelStream的地方实际上是公用的同一个ForkJoinPool。parallelStream提供了更简单的并发执行的实现，但并不意味着更高的性能，它是使用要根据具体的应用场景。如果cpu资源紧张parallelStream不会带来性能提升；如果存在频繁的线程切换反而会降低性能。

Stream总结

1. 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
2. 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
3. 所有 Stream 的操作必须以 lambda 表达式为参数
4. 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。
5. 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。