Java 8 Stream API 入门者教程

Java 8 给我们带来了一个新功能，也就是本文要介绍的 Stream API，它可以让我们以一种声明的方式处理数据。Stream 使用一种类似用 SQL 的语法来提供一种对 Java 集合运算和表达的高阶抽象。极大提高 Java 程序员的生产力，让程序员写出高效率、干净、简洁的代码。本文是 Java 8 Stream API 入门序列文章第一篇，将带领大家快速入门 Java 8 Stream API，当然，如果有 Scala 的使用基础，本文所涉及的 API 应该还是能够快速理解使用的。

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众帐号：iteblog_hadoop

Stream 创建

有很多方法可以创建不同源的 stream 实例，stream 实例一旦创建，将不会修改其源，因此我们从单个源创建多个 stream 实例。

Empty Stream

如果我们想创建一个空的 Stream，可以使用 empty() 方法，具体如下：

```
Stream<String> iteblogEmptyStream = Stream.empty();
```

通常在 streams 没有元素然后不想返回 null 的情况下使用：

```
public Stream<String> streamOf(List<String> list) {
   return list == null || list.isEmpty() ? Stream.empty() : list.stream();
}
```

通过集合（Collection）创建 Stream

Java 中的任何继承 Collection 接口的类都可以创建 Stream

```
List<String> list = Lists.newArrayList("iteblog", "iteblog_hadoop");
Stream<String> listStream = list.stream();

Set<String> set = Sets.newHashSet();
Stream<String> setStream = set.stream();
```

通过数组（Array）创建 Stream

数组也可以创建 Stream

```
Stream<String> streamOfArray = Stream.of("a", "b", "c");
```

当然，我们也可以通过已有的数组来创建 Stream

```
String[] iteblogArr = new String[]{"iteblog", "iteblog_hadoop", "java 8"};
Stream<String> streamOfArrayFull = Arrays.stream(iteblogArr);
Stream<String> streamOfArrayPart = Arrays.stream(iteblogArr, 1, 3);
```

通过 Stream.builder() 创建 Stream

Stream 提供了 builder 方法来创建 Stream：

```
Stream streamBuilder = Stream.builder().add("iteblog").add("iteblog_hadoop").add("java").build();
Stream<Object> streamBuilder = Stream.builder().add("iteblog").add("iteblog_hadoop").add("java").build();
```

上面创建的 Stream 类型是 Stream<Object>，如果我们想创建指定类型的 Stream，需要显示地指定类型

```
Stream<String> streamBuilder = Stream.<String>builder().add("iteblog").add("iteblog_hadoop").add("java").build();
```

通过 Stream.generate() 创建 Stream

Stream.generate() 方法接收一个 Supplier<T> 类型的参数来生成元素，生成的 stream 大小是无限的，所以我们需要指定 stream 生成的大小，以免出现内存不够的问题：

```
Stream<String> streamGenerated = Stream.generate(() -> "iteblog").limit(88);
```

通过 Stream.iterate() 创建 Stream

我们也可以通过 Stream.iterate() 来创建 Stream

```
Stream<Integer> streamIterated = Stream.iterate(2, n -> n * 2).limit(88);
```

Stream.iterate 方法的第一个参数将是这个 Stream 的第一个值，第二个元素将是前一个元素乘以 2。和 Stream.generate() 方法一样，我们也需要指定 stream 生成的大小，以免出现内存不够的问题。

通过原子类型创建 Stream

Java 8 中的 int, long 和 double 三个原子类型可以用来创建 streams，对应的接口分别是 IntStream, LongStream, DoubleStream

```
IntStream intStream = IntStream.range(0, 10);
```

```
LongStream longStream = LongStream.rangeClosed(0, 10);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0);
```

range(int startInclusive, int endExclusive) 相当于下面的代码：

```
for (long i = startInclusive; i < endExclusive ; i++) { ... }
```

rangeClosed(int startInclusive, int endInclusive) 相当于下面的代码：

```
for (long i = startInclusive; i <= endInclusive ; i++) { ... }
```

区别大家应该看出来了：rangeClosed 生成的 Stream 包含最后一个元素，而 range 缺不是。
当然，Java 8 中的 Random 类也为我们添加了生成上面三个原子类型对应的 Stream：

```
Random random = new Random();
IntStream intStream = random.ints(10);
LongStream longs = random.longs(10);
DoubleStream doubleStream = random.doubles(10);
```

通过字符串创建 Stream

Java 8 中的 String 类提供了 chars() 方法来创建 Stream：

```
IntStream streamOfChars = "abc".chars();
```

我们也可以通过下面方法来创建 Stream：

```
Stream<String> streamOfString = Pattern.compile(", ").splitAsStream("a, b, c");
```

通过文件创建 Stream

Java 8 的 Java NIO 类中 Files 允许我们通过 lines() 方法创建
Stream<String>，文件中的每一行数据将变成 stream 中的一个元素：

```
Path path = Paths.get("/user/iteblog/test.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset = Files.lines(path, Charset.forName("UTF-8"));
```

Stream 的引用

下面的代码是允许的：

```
Stream<String> stream = Stream.of("iteblog", "iteblog_hadoop", "spark")
            .filter(element -> element.contains("iteblog"));
Optional<String> anyElement = stream.findAny();
```

我们使用 stream 变量来引用一个定义好的 Stream，这个是允许的，接下来我们使用 findAny()
来操作这个 Stream，也是可以运行的。但是如果我们复用 stream
变量，这样执行的时候就会出现 IllegalStateException 异常：

```
Optional<String> firstElement = stream.findFirst();

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated
upon or closed
 at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
 at java.util.stream.ReferencePipeline.findFirst(ReferencePipeline.java:464)
 at com.java.iteblog.Java8Test.main(Java8Test.java:19)
```

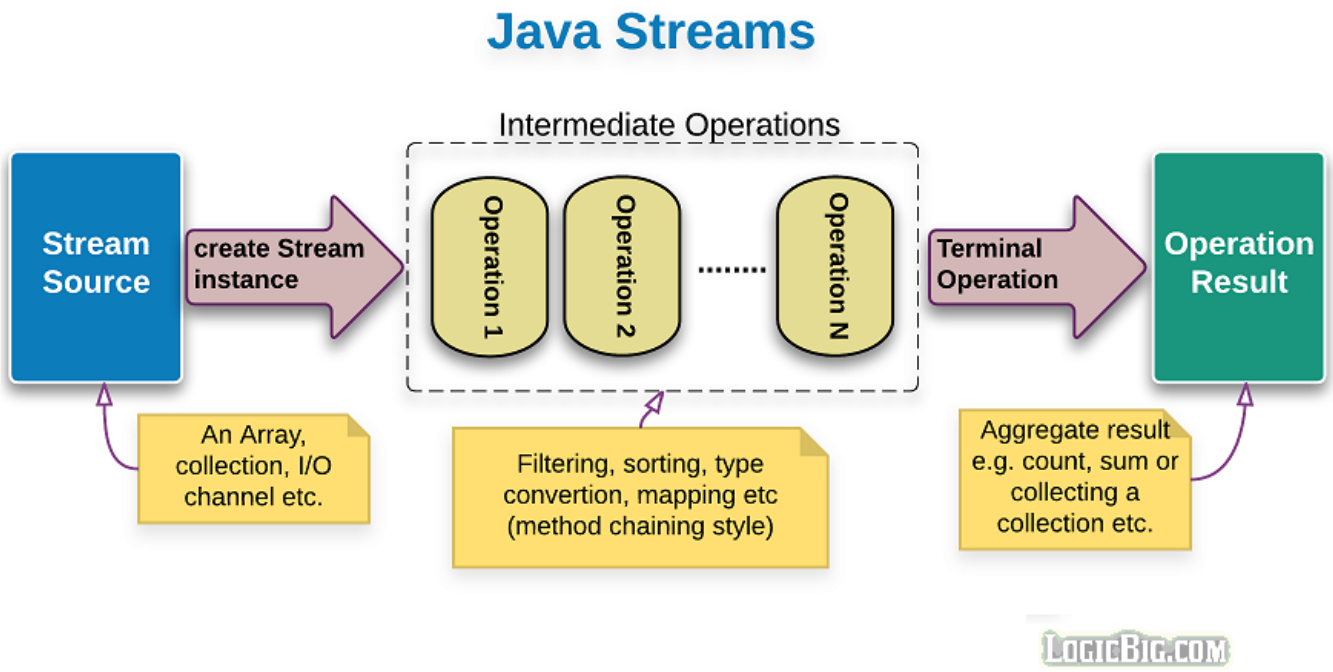
上面的实例说明 Java 8 streams 的引用是不可复用的。之所以这样是因为 Java 的 streams 设计目的是提供一种能力，以函数的方式将有限的操作序列应用于元素的源，而不是存储元素。如果我们这样写是可以的

```
List<String> iteblogList = Stream.of("iteblog", "iteblog_hadoop", "spark")
            .filter(element -> element.contains("iteblog")).collect(Collectors.toList());
```

```
Optional<String> anyElement = iteblogList.stream().findAny();
Optional<String> firstElement = iteblogList.stream().findFirst();
```

Stream 管道（stream pipeline）

要对数据源的元素执行一
系列操作并聚合它们的结果，需要三个部分：数据源（source）、中间操作（intermediate operations）和终端操作（terminal operation）。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众帐号：iteblog_hadoop

中间操作返回一个新的修改过的 Stream。比如下面的例子我们使用 skip() 方法跳过了旧 Stream 的第一个元素，并返回一个名为 iteblogSkip 的新 Stream：

```
Stream<String> iteblogSkip = Stream.of("iteblog", "iteblog_hadoop", "spark").skip(1);
```

如果需要多个修改操作，可以链接多个中间操作：

```
Stream<String> iteblogSkip = Stream.of("iteblog", "iteblog_hadoop", "spark")
```

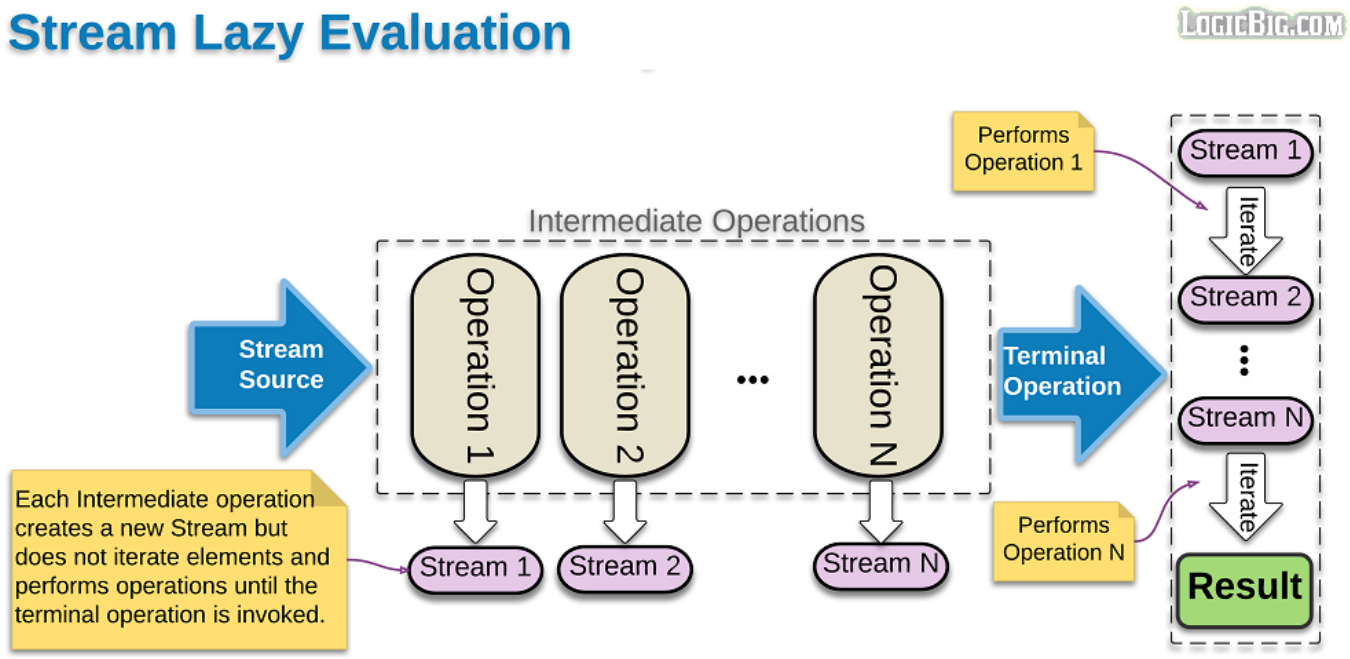
```
        .skip(1).map(element -> element.substring(0, 3));
```

上面例子我们同时使用了 skip() 和 map() 方法，并得到了有新的 Stream 引用。

stream 本身是没有价值的，用户真正感兴趣的是终端操作的结果，它可以是某种类型的值，也可以是应用于流的每个元素的操作。每个流只能使用一个终端操作。使用 stream
的正确和最方便的方式是通过 stream
管道，它是一个数据源、中间操作和终端操作的链（chain），例如:

```
long count = Stream.of("iteblog", "iteblog_hadoop", "spark")
        .skip(1).map(element -> element.substring(0, 3)).count();
```

延迟调用（Lazy Invocation）



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众帐号：iteblog_hadoop

中间操作是惰性（Lazy）的，这意味着仅在终端操作执行需要时才调用它们。为了说明这个，假设我们有一个名为 wasCalled() 的方法，每次被调用其内部的计数器都会递增：

```
private static long counter;
```

```
private static void wasCalled() {
    counter++;
}
```

现在我们在 filter() 中间操作中调用 wasCalled() 方法：

```
List<String> list = Arrays.asList("iteblog", "iteblog_hadoop", "spark");
counter = 0;
Stream<String> stream = list.stream().filter(element -> {
        wasCalled();
        return element.contains("iteblog");
});

System.out.println(counter);
```

我们的数据源 list 里面有三个元素，然后我们在这个数据源上调用了 filter() 方法，按道理应该会对每个元素调用一次 filter() 方法，这样 counter 比变量的值应该是 3。但是如果我们运行上面的代码你会发现 counter 的值还是 0！也就是说 filter() 方法根本就没调用，原因就是中间操作是惰性（Lazy）的，只有加上终端操作才会执行这个 filter() 方法。

我们把上面的代码修改成下面的代码：

```
List<String> list = Arrays.asList("iteblog", "iteblog_hadoop", "spark");

list.stream().filter(element -> {
        System.out.println("filter() was called");
        return element.contains("hadoop");
}).map(element -> {
        System.out.println("map() was called");
        return element.toUpperCase();
}).findFirst();
```

输出：

```
filter() was called
filter() was called
map() was called
```

可以看出，在终端输出了两次 filter() was called，一次 map() was called。也就是说 filter() 函数被调用了两次；map() 函数被调用了一次。Stream 管道是垂直执行的，在我们的例子里面，先运行 filter() 再运行 map()，只有 filter() 返回为 true 才会调用 map()，然后 findFirst() 只需要找到第一个满足的元素就可以终止程序的运行。

Stream 的运行顺序

从性能的角度来看，Stream 管道中不同操作的链接顺序是很重要的。下面两段代码片段的运行结果是一样的，但是下面的代码是推荐使用的。

```
long size = list.stream().map(element -> {
    wasCalled();
    return element.substring(0, 3);
}).skip(2).count();

long size = list.stream().skip(2).map(element -> {
    wasCalled();
    return element.substring(0, 3);
}).count();
```

因为第一个代码片段运行了三次 map()，而第二个代码片段只运行了一次 map()。所以在写 Java Stream 程序的时候，推荐 Stream 管道的顺序是：skip() -> filter() -> distinct()。

Stream 聚合

Stream API 有许多终端操作，它们将 Stream 聚合为一个原子类型的数据，例如 count()、max()、min()、sum() 等，但是这些操作根据预定义的实现进行工作的。如果开发人员需要定制 Stream 的聚合逻辑，该怎么办？这就是本小结要介绍的 reduce() 和 collect() 方法。

reduce() 方法介绍

reduce() 有三个重载的方法，但是都是接收以下几种类型的参数：

- identity：累加器的初始值，如果 stream 为空且没有要累加的内容，则为默认值；
- accumulator：指定元素聚合逻辑的函数。accumulator 在对数据源里面的每个元素进行聚合时，都会生成一个新的临时对象，生成的对象数等于数据源里面的元素个数，但是只有最后一个值是有用的，这个对性能不是很好的。
- combiner：聚合累加器结果的函数。只有在并行模式下才会调用 Combiner，以减少来自不同线程的累加器的结果。

现在让我们来看看如何来使用 reduce() 三个方法：

实例一

```
OptionalInt sum = IntStream.range(1, 4).reduce((a, b) -> a + b);
System.out.println(sum);
```

输出：
OptionalInt[6]（也就是 1 + 2 + 3）

实例二

```
int reducedTwoParams = IntStream.range(1, 4).reduce(10, (a, b) -> a + b);
System.out.println(reducedTwoParams);
```

输出：
16（也就是 10 + 1 + 2 + 3）

实例三

```
int reducedParams = Stream.of(1, 2, 3)
  .reduce(10, (a, b) -> a + b, (a, b) -> {
     System.out.println("combiner was called");
     return a + b;
  });

System.out.println(reducedParams);
```

输出：
16（也就是 10 + 1 + 2 + 3）

可以看出，实例三虽然我们指定了 combiner 但是控制台并没有输出 combiner was called，也就是说上面的 combiner 其实并没有调用。如果我们想调用 combiner，可以修改如下：

```
int reducedParallel = Arrays.asList(1, 2, 3).parallelStream()
    .reduce(10, (a, b) -> a + b, (a, b) -> {
       System.out.println("combiner was called");
```

```
    return a + b;
  });

System.out.println(reducedParallel);
```

输出：

```
combiner was called
combiner was called
36
```

可以看出这次的输出结果是 36，并且输出两次 combiner was called。之所以是 36，是因为上面程序先对每个元素调用 accumulator，也就是调用了三次 accumulator，然后其和累加器的初始值进行相加，因为这个 actions 是并行执行的，所以调用三次 accumulator 得到的结果为 (10 + 1 = 11; 10 + 2 = 12; 10 + 3 = 13) 。现在我们调用 combiner 把上面三次结果相加 (12 + 13 = 25; 25 + 11 = 36) 所以得到了 36。

collect() 方法介绍

collect() 方法也提供了聚合相关的逻辑实现，其函数签名为 R collect(Collector collector)，Java 8 中提供了大多数常用的 collectors 逻辑实现，我们可以直接使用。为了说明如何使用，我们还是提供一些例子：

```
static class Product {
    private int price;
    private String name;

    Product(int price, String name) {
        this.price = price;
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public String getName() {
        return name;
    }
```

```
        public void setName(String name) {
            this.name = name;
        }
}

List<Product> productList = Arrays.asList(new Product(23, "potatoes"),
  new Product(14, "orange"), new Product(13, "lemon"),
  new Product(23, "bread"), new Product(13, "sugar"));
```

将 productList 里面的商品名称全部拿出来，并转换成 list：

```
List<String> collectorCollection = productList.stream().map(Product::getName).collect(Collecto
rs.toList());
```

将 productList 里面的商品名称全部拿出来，并组合成 string：

```
String listToString = productList.stream().map(Product::getName)
                .collect(Collectors.joining(", ", "[", "]"));
```

计算 productList 里面所有商品的平均价格：

```
double averagePrice = productList.stream().collect(Collectors.averagingInt(Product::getPrice));
```

计算 productList 里面所有商品的总价：

```
int summingPrice = productList.stream().collect(Collectors.summingInt(Product::getPrice));
```

计算 productList 里面所有商品统计信息：

```
IntSummaryStatistics statistics = productList.stream().collect(Collectors.summarizingInt(Produ
```

```
ct::getPrice));
System.out.println(statistics);
```

输出：

```
IntSummaryStatistics{count=5, sum=86, min=13, average=17.200000, max=23}
```

按照商品价格对商品进行归类

```
Map<Integer, List<Product>> collectorMapOfLists = productList.stream()
            .collect(Collectors.groupingBy(Product::getPrice));
```

上面的结果是价格相等的商品都放到同一个 List<Product> 中。
按照相关逻辑对商品进行分组

```
Map<Boolean, List<Product>> mapPartioned = productList.stream()
  .collect(Collectors.partitioningBy(element -> element.getPrice() > 15));
```

上面程序的运行结果是价格大于15的放到一个 List<Product> 中。
将 list 转换成 set

```
Set<Product> unmodifiableSet = productList.stream()
  .collect(Collectors.collectingAndThen(Collectors.toSet(),
  Collections::unmodifiableSet));
```

自定义 collector
总有些原因系统自带的 API 无法满足我们的需求，这时候我们就可以自定义 collector。比如下面我们自定义了一个 collector，把所有的商品放到 LinkedList<Product> 里面：

```
Collector<Product, ?, LinkedList<Product>> toLinkedList =
  Collector.of(LinkedList::new, LinkedList::add,
    (first, second) -> {
      first.addAll(second);
```

```
    return first;
});

LinkedList<Product> linkedListOfPersons = productList.stream().collect(toLinkedList);
```

总结

Stream API 是一套功能强大但易于理解的用于处理元素序列的工具。它允许我们减少大量的代码，创建更多可读的程序，并可以提高应用程序的生产率。本博客将在后面几篇文章介绍如何更好地使用 Java 8 Stream API。