

加餐2 | 带你吃透课程中Java 8的那些重要知识点（下）

2020-03-17 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



讲述：王少泽

时长 14:27 大小 13.24M



你好，我是朱晔。

上一讲的几个例子中，其实都涉及了 Stream API 的最基本使用方法。今天，我会与你详细介绍复杂、功能强大的 Stream API。

Stream 流式操作，用于对集合进行投影、转换、过滤、排序等，更进一步地，这些操作能链式串联在一起使用，类似于 SQL 语句，可以大大简化代码。可以说，Stream 操作是 Java 8 中最重要的内容，也是这个课程大部分代码都会用到的操作。



我先说明下，有些案例可能不太好理解，建议你对着代码逐一到源码中查看 Stream 操作的方法定义，以及 JDK 中的代码注释。

Stream 操作详解

为了方便你理解 Stream 的各种操作，以及后面的案例，我先把这节课涉及的 Stream 操作汇总到了一张图中。你可以先熟悉一下。

方法	中文	操作类型	类比SQL	使用的类型/函数式接口	作用
filter	筛选/过滤	中间	where	Predicate<T>	对流过滤，使元素符合传入条件
map	转换/投影	中间	select	Function<T,R>	使用传入的函数，对流中每一个元素进行转换
flatMap	展开/扁平化	中间	N/A	Function<T,Stream<R>>	相当于map+flat，通过map把每一个元素转换为一个流，然后把所有流链接到一起扁平化展开
sorted	排序	中间	order by	Comparator<T>	使用传入的比较器，对流中的元素排序
distinct	去重	中间	distinct	long	对流中元素去重（使用Object.equals判重）
skip & limit	分页	中间	limit	long	跳过流中部分元素以及限制元素数量
collect	收集	终结	N/A	Collector<T, A, R>	对流进行终结操作，把流导出成为我们需要的数据结构
forEach	遍历	终结	N/A	Consumer<T>	对每一个元素遍历进行消费
anyMatch	是否有元素匹配	终结	N/A	Predicate<T>	使用谓词判断是否有任何一个元素满足匹配
allMatch	是否所有元素匹配	终结	N/A	Predicate<T>	使用谓词判断是否所有元素都满足匹配

在接下来的讲述中，我会围绕订单场景，给出如何使用 Stream 的各种 API 完成订单的统计、搜索、查询等功能，和你一起学习 Stream 流式操作的各种方法。你可以结合代码中的注释理解案例，也可以自己运行源码观察输出。

我们先定义一个订单类、一个订单商品类和一个顾客类，用作后续 Demo 代码的数据结构：

复制代码

```
1 //订单类
2 @Data
3 public class Order {
4     private Long id;
5     private Long customerId;//顾客ID
6     private String customerName;//顾客姓名
```

```

7     private List<OrderItem> orderItemList;//订单商品明细
8     private Double totalPrice;//总价格
9     private LocalDateTime placedAt;//下单时间
10 }
11 //订单商品类
12 @Data
13 @AllArgsConstructor
14 @NoArgsConstructor
15 public class OrderItem {
16     private Long productId;//商品ID
17     private String productName;//商品名称
18     private Double productPrice;//商品价格
19     private Integer productQuantity;//商品数量
20 }
21 //顾客类
22 @Data
23 @AllArgsConstructor
24 public class Customer {
25     private Long id;
26     private String name;//顾客姓名
27 }

```

在这里，我们有一个 orders 字段保存了一些模拟数据，类型是 List。这里，我就不贴出生成模拟数据的代码了。这不会影响你理解后面的代码，你也可以自己下载源码阅读。

创建流

要使用流，就要先创建流。创建流一般有五种方式：

通过 stream 方法把 List 或数组转换为流；

通过 Stream.of 方法直接传入多个元素构成一个流；

通过 Stream.iterate 方法使用迭代的方式构造一个无限流，然后使用 limit 限制流元素个数；

通过 Stream.generate 方法从外部传入一个提供元素的 Supplier 来构造无限流，然后使用 limit 限制流元素个数；

通过 IntStream 或 DoubleStream 构造基本类型的流。

 复制代码

```

1 //通过stream方法把List或数组转换为流
2 @Test
3 public void stream()

```

```

4 {
5     Arrays.asList("a1", "a2", "a3").stream().forEach(System.out::println);
6     Arrays.stream(new int[]{1, 2, 3}).forEach(System.out::println);
7 }
8
9 //通过Stream.of方法直接传入多个元素构成一个流
10 @Test
11 public void of()
12 {
13     String[] arr = {"a", "b", "c"};
14     Stream.of(arr).forEach(System.out::println);
15     Stream.of("a", "b", "c").forEach(System.out::println);
16     Stream.of(1, 2, "a").map(item -> item.getClass().getName()).forEach(System
17 }
18
19 //通过Stream.iterate方法使用迭代的方式构造一个无限流, 然后使用limit限制流元素个数
20 @Test
21 public void iterate()
22 {
23     Stream.iterate(2, item -> item * 2).limit(10).forEach(System.out::println)
24     Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.TEN)).limit(10).forE
25 }
26
27 //通过Stream.generate方法从外部传入一个提供元素的Supplier来构造无限流, 然后使用limit限制
28 @Test
29 public void generate()
30 {
31     Stream.generate(() -> "test").limit(3).forEach(System.out::println);
32     Stream.generate(Math::random).limit(10).forEach(System.out::println);
33 }
34
35 //通过IntStream或DoubleStream构造基本类型的流
36 @Test
37 public void primitive()
38 {
39     //演示IntStream和DoubleStream
40     IntStream.range(1, 3).forEach(System.out::println);
41     IntStream.range(0, 3).mapToObj(i -> "x").forEach(System.out::println);
42
43     IntStream.rangeClosed(1, 3).forEach(System.out::println);
44     DoubleStream.of(1.1, 2.2, 3.3).forEach(System.out::println);
45
46     //各种转换, 后面注释代表了输出结果
47     System.out.println(IntStream.of(1, 2).toArray().getClass()); //class [I
48     System.out.println(Stream.of(1, 2).mapToInt(Integer::intValue).toArray().g
49     System.out.println(IntStream.of(1, 2).boxed().toArray().getClass()); //cla
50     System.out.println(IntStream.of(1, 2).asDoubleStream().toArray().getClass(
51     System.out.println(IntStream.of(1, 2).asLongStream().toArray().getClass())
52
53     //注意基本类型流和装箱后的流的区别
54     Arrays.asList("a", "b", "c").stream() // Stream<String>
55         .mapToInt(String::length) // IntStream

```

```

56         .asLongStream()                // LongStream
57         .mapToDouble(x -> x / 10.0)    // DoubleStream
58         .boxed()                       // Stream<Double>
59         .mapToLong(x -> 1L)            // LongStream
60         .mapToObj(x -> "")             // Stream<String>
61         .collect(Collectors.toList());
62     }

```


filter

filter 方法可以实现过滤操作，类似 SQL 中的 where。我们可以使用一行代码，通过 filter 方法实现查询所有订单中最近半年金额大于 40 的订单，通过连续叠加 filter 方法进行多次条件过滤：

```

1  //最近半年的金额大于40的订单
2  orders.stream()
3      .filter(Objects::nonNull) //过滤null值
4      .filter(order -> order.getPlacedAt().isAfter(LocalDate.now().minus(
5          .filter(order -> order.getTotalPrice() > 40) //金额大于40的订单
6          .forEach(System.out::println);

```

 复制代码

如果不使用 Stream 的话，必然需要一个中间集合来收集过滤后的结果，而且所有的过滤条件会堆积在一起，代码冗长且不易读。

map

map 操作可以做转换（或者说投影），类似 SQL 中的 select。为了对比，我用两种方式统计订单中所有商品的数量，前一种是通过两次遍历实现，后一种是通过两次 mapToLong+sum 方法实现：

```

1  //计算所有订单商品数量
2  //通过两次遍历实现
3  LongAdder longAdder = new LongAdder();
4  orders.stream().forEach(order ->
5      order.getOrderItemList().forEach(orderItem -> longAdder.add(orderItem.);
6
7  //使用两次mapToLong+sum方法实现
8  assertThat(longAdder.longValue(), is(orders.stream().mapToLong(order ->
9      order.getOrderItemList().stream()

```

 复制代码

显然，后一种方式无需中间变量 `longAdder`，更直观。

这里再补充一下，使用 `for` 循环生成数据，是我们平时常用的操作，也是这个课程会大量用到的。现在，我们可以用一行代码使用 `IntStream` 配合 `mapToObj` 替代 `for` 循环来生成数据，比如生成 10 个 `Product` 元素构成 `List`：

[复制代码](#)

```
1 //把IntStream通过转换Stream<Project>
2 System.out.println(IntStream.rangeClosed(1,10)
3     .mapToObj(i->new Product((long)i, "product"+i, i*100.0))
4     .collect(toList()));
```

flatMap

接下来，我们看看 `flatMap` 展开或者叫扁平化操作，相当于 `map+flat`，通过 `map` 把每一个元素替换为一个流，然后展开这个流。

比如，我们要统计所有订单的总价格，可以有两种方式：

直接通过原始商品列表的商品个数 * 商品单价统计的话，可以先把订单通过 `flatMap` 展开成商品清单，也就是把 `Order` 替换为 `Stream`，然后对每一个 `OrderItem` 用 `mapToDouble` 转换获得商品总价，最后进行一次 `sum` 求和；

利用 `flatMapToDouble` 方法把列表中每一项展开替换为一个 `DoubleStream`，也就是直接把每一个订单转换为每一个商品的总价，然后求和。

[复制代码](#)

```
1 //直接展开订单商品进行价格统计
2 System.out.println(orders.stream()
3     .flatMap(order -> order.getOrderItemList().stream())
4     .mapToDouble(item -> item.getProductQuantity() * item.getProductPrice())
5     .sum());
6 //另一种方式flatMap+mapToDouble=flatMapToDouble
7 System.out.println(orders.stream()
8     .flatMapToDouble(order ->
9         order.getOrderItemList()
10            .stream().mapToDouble(item -> item.getProductQuantity() * item.getProductPrice())
11     ).sum());
```

```
11         .sum());
```

这两种方式可以得到相同的结果，并无本质区别。

sorted

sorted 操作可以用于行内排序的场景，类似 SQL 中的 order by。比如，要实现大于 50 元订单的按价格倒序取前 5，可以通过 Order::getTotalPrice 方法引用直接指定需要排序的依据字段，通过 reversed() 实现倒序：

 复制代码


```
1 //大于50的订单,按照订单价格倒序前5
2 orders.stream().filter(order -> order.getTotalPrice() > 50)
3     .sorted(comparing(Order::getTotalPrice).reversed())
4     .limit(5)
5     .forEach(System.out::println);
```

distinct

distinct 操作的作用是去重，类似 SQL 中的 distinct。比如下面的代码实现：

查询去重后的下单用户。使用 map 从订单提取出购买用户，然后使用 distinct 去重。

查询购买过的商品名。使用 flatMap+map 提取出订单中所有的商品名，然后使用 distinct 去重。

 复制代码

```
1 //去重的下单用户
2 System.out.println(orders.stream().map(order -> order.getCustomerName()).distinct().count());
3
4 //所有购买过的商品
5 System.out.println(orders.stream()
6     .flatMap(order -> order.getOrderItemList().stream())
7     .map(OrderItem::getProductName)
8     .distinct().collect(joining(", ")));
```

skip & limit

skip 和 limit 操作用于分页，类似 MySQL 中的 limit。其中，skip 实现跳过一定的项，limit 用于限制项总数。比如下面的两段代码：

按照下单时间排序，查询前 2 个订单的顾客姓名和下单时间；

按照下单时间排序，查询第 3 和第 4 个订单的顾客姓名和下单时间。

 复制代码

```
1 //按照下单时间排序，查询前2个订单的顾客姓名和下单时间
2 orders.stream()
3     .sorted(comparing(Order::getPlacedAt))
4     .map(order -> order.getCustomerName() + "@" + order.getPlacedAt())
5     .limit(2).forEach(System.out::println);
6 //按照下单时间排序，查询第3和第4个订单的顾客姓名和下单时间
7 orders.stream()
8     .sorted(comparing(Order::getPlacedAt))
9     .map(order -> order.getCustomerName() + "@" + order.getPlacedAt())
10    .skip(2).limit(2).forEach(System.out::println);
```

collect

collect 是收集操作，对流进行终结（终止）操作，把流导出为我们需要的数据结构。“终结”是指，导出后，无法再串联使用其他中间操作，比如 filter、map、flatMap、sorted、distinct、limit、skip。

在 Stream 操作中，collect 是最复杂的终结操作，比较简单的终结操作还有 forEach、toArray、min、max、count、anyMatch 等，我就不再展开了，你可以查询 [JDK 文档](#)，搜索 terminal operation 或 intermediate operation。

接下来，我通过 6 个案例，来演示下几种比较常用的 collect 操作：

第一个案例，实现了字符串拼接操作，生成一定位数的随机字符串。

第二个案例，通过 Collectors.toSet 静态方法收集为 Set 去重，得到去重后的下单用户，再通过 Collectors.joining 静态方法实现字符串拼接。

第三个案例，通过 Collectors.toCollection 静态方法获得指定类型的集合，比如把 List 转换为 LinkedList。

第四个案例，通过 `Collectors.toMap` 静态方法将对象快速转换为 Map，Key 是订单 ID、Value 是下单用户名。

第五个案例，通过 `Collectors.toMap` 静态方法将对象转换为 Map。Key 是下单用户名，Value 是下单时间，一个用户可能多次下单，所以直接在这里进行了合并，只获取最近一次的下单时间。

第六个案例，使用 `Collectors.summingInt` 方法对商品数量求和，再使用 `Collectors.averagingInt` 方法对结果求平均值，以统计所有订单平均购买的商品数量。

 复制代码

```
1 //生成一定位数的随机字符串
2 System.out.println(random.ints(48, 122)
3     .filter(i -> (i < 57 || i > 65) && (i < 90 || i > 97))
4     .mapToObj(i -> (char) i)
5     .limit(20)
6     .collect(StringBuilder::new, StringBuilder::append, StringBuilder::append)
7     .toString());
8
9 //所有下单的用户，使用toSet去重后实现字符串拼接
10 System.out.println(orders.stream()
11     .map(order -> order.getCustomerName()).collect(toSet())
12     .stream().collect(joining(", ", "[", "]")));
13
14 //用toCollection收集器指定集合类型
15 System.out.println(orders.stream().limit(2).collect(toCollection(LinkedList::new)));
16
17 //使用toMap获取订单ID+下单用户名的Map
18 orders.stream()
19     .collect(toMap(Order::getId, Order::getCustomerName))
20     .entrySet().forEach(System.out::println);
21
22 //使用toMap获取下单用户名+最近一次下单时间的Map
23 orders.stream()
24     .collect(toMap(Order::getCustomerName, Order::getPlacedAt, (x, y) -> x.isAfter(y)))
25     .entrySet().forEach(System.out::println);
26
27 //订单平均购买的商品数量
28 System.out.println(orders.stream().collect(averagingInt(order ->
29     order.getOrderItemList().stream()
30         .collect(summingInt(OrderItem::getProductQuantity)))));
```

可以看到，这 6 个操作使用 Stream 方式一行代码就可以实现，但使用非 Stream 方式实现的话，都需要几行甚至十几行代码。

有关 Collectors 类的一些常用静态方法，我总结到了一张图中，你可以再整理一下思路：

方法	返回类型	作用
toList	List<T>	把流中的元素收集成为一个List
toSet	Set<T>	把流中的元素收集成为一个Set，去重
toCollection	Collection<T>	把流中的元素收集成为指定的集合
counting	Long	计算流中的元素个数
summingInt	Integer	对流中元素的某个整数属性求和
averagingInt	Double	对流中元素的某个整数属性求平均值
joining	String	连接流中元素toString后的字符串
minBy	Optional<T>	使用指定的比较器选出最小元素
maxBy	Optional<T>	使用指定的比较器选出最大元素
collectingAndThen	根据收集器的返回	包裹另一个收集器，对结果进行转换
groupBy	Map<K,List<T>>>	根据元素的一个属性值对元素分组，属性值作为Key
partitionBy	Map<Boolean,List<T>>>	根据流中元素应用谓词（Predicate）的结果，将元素分成true和false两个区

其中，groupBy 和 partitionBy 比较复杂，我和你举例介绍。

groupBy

groupBy 是分组统计操作，类似 SQL 中的 group by 子句。它和后面介绍的 partitioningBy 都是特殊的收集器，同样也是终结操作。分组操作比较复杂，为帮你理解得更透彻，我准备了 8 个案例：

- 第一个案例，按照用户名分组，使用 Collectors.counting 方法统计每个人的下单数量，再按照下单数量倒序输出。
- 第二个案例，按照用户名分组，使用 Collectors.summingDouble 方法统计订单总金额，再按总金额倒序输出。

第三个案例，按照用户名分组，使用两次 `Collectors.summingInt` 方法统计商品采购数量，再按总数量倒序输出。


第四个案例，统计被采购最多的商品。先通过 `flatMap` 把订单转换为商品，然后把商品名作为 Key、`Collectors.summingInt` 作为 Value 分组统计采购数量，再按 Value 倒序获取第一个 Entry，最后查询 Key 就得到了售出最多的商品。

第五个案例，同样统计采购最多的商品。相比第四个案例排序 Map 的方式，这次直接使用 `Collectors.maxBy` 收集器获得最大的 Entry。

第六个案例，按照用户名分组，统计用户下的金额最高的订单。Key 是用户名，Value 是 Order，直接通过 `Collectors.maxBy` 方法拿到金额最高的订单，然后通过 `collectingAndThen` 实现 `Optional.get` 的内容提取，最后遍历 Key/Value 即可。

第七个案例，根据下单年月分组统计订单 ID 列表。Key 是格式化成年月后的下单时间，Value 直接通过 `Collectors.mapping` 方法进行了转换，把订单列表转换为订单 ID 构成的 List。

第八个案例，根据下单年月 + 用户名两次分组统计订单 ID 列表，相比上一个案例多了一次分组操作，第二次分组是按照用户名进行分组。

 复制代码

```
1 //按照用户名分组，统计下单数量
2 System.out.println(orders.stream().collect(groupingBy(Order::getCustomerName,
3     .entrySet().stream().sorted(Map.Entry.<String, Long>comparingByValue())
4
5 //按照用户名分组，统计订单总金额
6 System.out.println(orders.stream().collect(groupingBy(Order::getCustomerName,
7     .entrySet().stream().sorted(Map.Entry.<String, Double>comparingByValue
8
9 //按照用户名分组，统计商品采购数量
10 System.out.println(orders.stream().collect(groupingBy(Order::getCustomerName,
11     summingInt(order -> order.getOrderItemList().stream()
12         .collect(summingInt(OrderItem::getProductQuantity))))))
13     .entrySet().stream().sorted(Map.Entry.<String, Integer>comparingByValue
14
15 //统计最受欢迎的商品，倒序后取第一个
16 orders.stream()
17     .flatMap(order -> order.getOrderItemList().stream())
18     .collect(groupingBy(OrderItem::getProductName, summingInt(OrderItem::g
19     .entrySet().stream()
20     .sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
21     .map(Map.Entry::getKey)
22     .findFirst()
23     .ifPresent(System.out::println);
24
```

```

25 //统计最受欢迎的商品的另一种方式，直接利用maxBy
26 orders.stream()
27     .flatMap(order -> order.getOrderItemList().stream())
28     .collect(groupingBy(OrderItem::getProductName, summingInt(OrderItem::g
29     .entrySet().stream()
30     .collect(maxBy(Map.Entry.comparingByValue()))
31     .map(Map.Entry::getKey)
32     .ifPresent(System.out::println);
33
34 //按照用户名分组，选用户下的总金额最大的订单
35 orders.stream().collect(groupingBy(Order::getCustomerName, collectingAndThen(m
36     .forEach((k, v) -> System.out.println(k + "#" + v.getTotalPrice() + "@
37
38 //根据下单年月分组，统计订单ID列表
39 System.out.println(orders.stream().collect
40     (groupingBy(order -> order.getPlacedAt().format(DateTimeFormatter.ofPa
41     mapping(order -> order.getId(), toList()))));
42
43 //根据下单年月+用户名两次分组，统计订单ID列表
44 System.out.println(orders.stream().collect
45     (groupingBy(order -> order.getPlacedAt().format(DateTimeFormatter.ofPa
46     groupingBy(order -> order.getCustomerName(),
47     mapping(order -> order.getId(), toList()))));

```

如果不借助 Stream 转换为普通的 Java 代码，实现这些复杂的操作可能需要几十行代码。

partitionBy

partitioningBy 用于分区，分区是特殊的分组，只有 true 和 false 两组。比如，我们把用户按照是否下单进行分区，给 partitioningBy 方法传入一个 Predicate 作为数据分区的区分，输出是 Map<Boolean, List>：

 复制代码

```

1 public static <T>
2 Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> pre
3     return partitioningBy(predicate, toList());
4 }

```

测试一下，partitioningBy 配合 anyMatch，可以把用户分为下过订单和没下过订单两组：

```
1 //根据是否有下单记录进行分区
2 System.out.println(Customer.getData().stream().collect(
3     partitioningBy(customer -> orders.stream().mapToLong(Order::getCustome
4         .anyMatch(id -> id == customer.getId()))));
```

重点回顾

今天，我用了大量的篇幅和案例，和你展开介绍了 Stream 中很多具体的流式操作方法。有些案例可能不太好理解，我建议你对着代码逐一到源码中查看这些操作的方法定义，以及 JDK 中的代码注释。

最后，我建议你思考下，在日常工作中还会使用 SQL 统计哪些信息，这些 SQL 是否也可以用 Stream 来改写呢？Stream 的 API 博大精深，但其中又有规律可循。这其中的规律主要就是，理清楚这些 API 传参的函数式接口定义，就能搞明白到底是需要我们提供数据、消费数据、还是转换数据等。那，掌握 Stream 的方法便是，多测试多练习，以强化记忆、加深理解。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

思考与讨论

1. 使用 Stream 可以非常方便地对 List 做各种操作，那有没有什么办法可以实现在整个过程中观察数据变化呢？比如，我们进行 filter+map 操作，如何观察 filter 后 map 的原始数据呢？
2. Collectors 类提供了很多现成的收集器，那我们有没有办法实现自定义的收集器呢？比如，实现一个 MostPopularCollector，来得到 List 中出现次数最多的元素，满足下面两个测试用例：

```
1 assertThat(Stream.of(1, 1, 2, 2, 2, 3, 4, 5, 5).collect(new MostPopularCollecto
2 assertThat(Stream.of('a', 'b', 'c', 'c', 'c', 'd').collect(new MostPopularColl
```

关于 Java 8，你还有什么使用心得吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 加餐1 | 带你吃透课程中Java 8的那些重要知识点（上）

精选留言 (3)

 写留言



张璐 置顶

2020-03-17

为了写作业又去翻了《Java 8 实战》，测试终于通过了

```
public class MostPopularCollector
```

```
    implements Collector<Object, // 收集String流
```

```
        Map<Object, Integer>, // 累加器是一个Map, key为字符, value为出现的次数
```

```
        Optional> // 返回的是出现次数最多的字符...
```

展开 

作者回复: 



Wiggle Wiggle 置顶

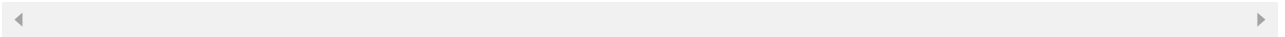
2020-03-17

Stream API 有个 peek 方法可以接收一个 consumer 来打印数据，可以接在任意 transfo

rmation 操作后面查看数据

展开 ▾

作者回复: 是的, 这是一种方法, 此外IDEA已经增加了非常方便的Stream调试功能, 可以参考<https://www.jetbrains.com/help/idea/analyze-java-stream-operations.html>



pedro

2020-03-17

我目前想到的数据观察的方式比较原始, 一种是通过log打印, 一种是debug。但我肯定这都不是啥好办法, 希望老师告知解放生产力的方法。

展开 ▾

作者回复: IDEA已经增加了非常方便的Stream调试功能, 可以参考<https://www.jetbrains.com/help/idea/analyze-java-stream-operations.html>

