链滴

《Java8 实战》- 第五章读书笔记（使用流 Stream-02）

作者：Not-Found
原文链接：https://ld246.com/article/1537078683550
来源网站：链滴

付诸实战

在本节中，我们会将迄今学到的关于流的知识付诸实践。我们来看一个不同的领域：执行交易的交易。你的经理让你为八个查询找到答案。

1. 找出2011年发生的所有交易，并按交易额排序（从低到高）。
2. 交易员都在哪些不同的城市工作过?
3. 查找所有来自于剑桥的交易员，并按姓名排序。
4. 返回所有交易员的姓名字符串，按字母顺序排序。
5. 有没有交易员是在米兰工作的?
6. 打印生活在剑桥的交易员的所有交易额。
7. 所有交易中，最高的交易额是多少?
8. 找到交易额最小的交易。

领域：交易员和交易

以下是我们要处理的领域，一个 Traders 和 Transactions 的列表：

```
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario", "Milan");
Trader alan = new Trader("Alan", "Cambridge");
Trader brian = new Trader("Brian", "Cambridge");

List<Transaction> transactions = Arrays.asList(
        new Transaction(brian, 2011, 300),
        new Transaction(raoul, 2012, 1000),
        new Transaction(raoul, 2011, 400),
        new Transaction(mario, 2012, 710),
        new Transaction(mario, 2012, 700),
        new Transaction(alan, 2012, 950)
);
```

Trader和Transaction类的定义：

```
public class Trader {
    private String name;
    private String city;

    public Trader(String n, String c){
        this.name = n;
        this.city = c;
    }

    public String getName() {
        return name;
    }

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

```
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    @Override
    public String toString() {
        return "Trader{" +
                "name='" + name + '\'' +
                ", city='" + city + '\'' +
                '}';
    }
}
```

Transaction类：

```
public class Transaction {
    private Trader trader;
    private Integer year;
    private Integer value;

    public Transaction(Trader trader, Integer year, Integer value) {
        this.trader = trader;
        this.year = year;
        this.value = value;
    }

    public Trader getTrader() {
        return trader;
    }

    public void setTrader(Trader trader) {
        this.trader = trader;
    }

    public Integer getYear() {
        return year;
    }

    public void setYear(Integer year) {
        this.year = year;
    }

    public Integer getValue() {
        return value;
    }

    public void setValue(Integer value) {
```

```
        this.value = value;
    }

    @Override
    public String toString() {
        return "Transaction{" +
                "trader=" + trader +
                ", year=" + year +
                ", value=" + value +
                '}';
    }
}
```

首先，我们来看第一个问题：找出2011年发生的所有交易，并按交易额排序（从低高）。

```
List<Transaction> tr2011 = transactions.stream()
                // 筛选出2011年发生的所有交易
                .filter(transaction -> transaction.getYear() == 2011)
                // 按照交易额从低到高排序
                .sorted(Comparator.comparing(Transaction::getValue))
                // 转为集合
                .collect(Collectors.toList());
```

太棒了，第一个问题我们很轻松的就解决了！首先，将transactions集合转为流，然后给filter传递一谓词来选择2011年的交易，接着按照交易额从低到高进行排序，最后将Stream中的所有元素收集到个List集合中。

第二个问题：交易员都在哪些不同的城市工作过?

```
List<String> cities = transactions.stream()
                // 提取出交易员所工作的城市
                .map(transaction -> transaction.getTrader().getCity())
                // 去除已有的城市
                .distinct()
                // 将Stream中所有的元素转为一个List集合
                .collect(Collectors.toList());
```

是的，我们很简单的完成了第二个问题。首先，将transactions集合转为流，然后使用map提取出与易员相关的每位交易员所在的城市，接着使用distinct去除重复的城市（当然，我们也可以去掉distinc，在最后我们就要使用collect，将Stream中的元素转为一个Set集合。collect(Collectors.toSet()))我们只需要不同的城市，最后将Stream中的所有元素收集到一个List中。

第三个问题：查找所有来自于剑桥的交易员，并按姓名排序。

```
List<Trader> traders = transactions.stream()
                // 从交易中提取所有的交易员
                .map(Transaction::getTrader)
                // 进选择位于剑桥的交易员
                .filter(trader -> "Cambridge".equals(trader.getCity()))
                // 确保没有重复
                .distinct()
```

```
        // 对生成的交易员流按照姓名进行排序
        .sorted(Comparator.comparing(Trader::getName))
        .collect(Collectors.toList());
```

第三个问题，从交易中提取所有的交易员，然后进选择位于剑桥的交易员确保没有重复，接着对生成交易员流按照姓名进行排序。

第四个问题：返回所有交易员的姓名字符串，按字母顺序排序。

```
String traderStr =
        transactions.stream()
            // 提取所有交易员姓名，生成一个 Strings 构成的 Stream
            .map(transaction -> transaction.getTrader().getName())
            // 只选择不相同的姓名
            .distinct()
            // 对姓名按字母顺序排序
            .sorted()
            // 逐个拼接每个名字，得到一个将所有名字连接起来的 String
            .reduce("", (n1, n2) -> n1 + " " + n2);
```

这些问题，我们都很轻松的就完成！首先，提取所有交易员姓名，生成一个 Strings 构成的 Stream 且只选择不相同的姓名，然后对姓名按字母顺序排序，最后使用reduce将名字拼接起来！

请注意，此解决方案效率不高（所有字符串都被反复连接，每次迭代的时候都要建立一个新的 String 对象）。下一章中，你将看到一个更为高效的解决方案，它像下面这样使用 joining （其内部会用到 StringBuilder ）：

```
String traderStr =
        transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .collect(joining());
```

第五个问题：有没有交易员是在米兰工作的?

```
boolean milanBased =
        transactions.stream()
            // 把一个谓词传递给 anyMatch ，检查是否有交易员在米兰工作
            .anyMatch(transaction -> "Milan".equals(transaction.getTrader()
                .getCity()));
```

第五个问题，依旧很简单把一个谓词传递给 anyMatch ，检查是否有交易员在米兰工作。

第六个问题：打印生活在剑桥的交易员的所有交易额。

```
transactions.stream()
        // 选择住在剑桥的交易员所进行的交易
        .filter(t -> "Cambridge".equals(t.getTrader().getCity()))
        // 提取这些交易的交易额
        .map(Transaction::getValue)
        // 打印每个值
```

```
        .forEach(System.out::println);
```

第六个问题，首先选择住在剑桥的交易员所进行的交易，接着提取这些交易的交易额，然后就打印出个值。

第七个问题：所有交易中，最高的交易额是多少?

```
Optional<Integer> highestValue =
        transactions.stream()
            // 提取每项交易的交易额
            .map(Transaction::getValue)
            // 计算生成的流中的最大值
            .reduce(Integer::max);
```

第七个问题，首先提取每项交易的交易额，然后使用reduce计算生成的流中的最大值。

第八个问题：找到交易额最小的交易。

```
Optional<Transaction> smallestTransaction =
        transactions.stream()
            // 通过反复比较每个交易的交易额，找出最小的交易
            .reduce((t1, t2) ->
                t1.getValue() < t2.getValue() ? t1 : t2);
```

是的，第八个问题很简单，但是还有更好的做法！流支持 min 和 max 方法，它们可以接受一个 Comparator 作为参数，指定

计算最小或最大值时要比较哪个键值：

```
Optional<Transaction> smallestTransaction = transactions.stream()
                        .min(comparing(Transaction::getValue));
```

上面的八个问题，我们通过Stream很轻松的就完成了，真是太棒了！

数值流

我们在前面看到了可以使用 reduce 方法计算流中元素的总和。例如，你可以像下面这样计

算菜单的热量：

```
int calories = menu.stream()
            .map(Dish::getCalories)
            .reduce(0, Integer::sum);
```

这段代码的问题是，它有一个暗含的装箱成本。每个 Integer 都必须拆箱成一个原始类型，

再进行求和。要是可以直接像下面这样调用 sum 方法，岂不是更好?

```
int calories = menu.stream()
            .map(Dish::getCalories)
            .sum();
```

但这是不可能的。问题在于 map 方法会生成一个 Stream<T> 。虽然流中的元素是 Integer 类

型，但 Streams 接口没有定义 sum 方法。为什么没有呢？比方说，你只有一个像 menu 那样的Stre

m<Dish>，把各种菜加起来是没有任何意义的。但不要担心，Stream API还提供了原始类型流特化专门支持处理数值流的方法。

原始类型流特化

Java 8引入了三个原始类型特化流接口来解决这个问题： IntStream 、 DoubleStream 和 LongStream ，分别将流中的元素特化为 int 、 long 和 double ，从而避免了暗含的装箱成本。每接口都带来了进行常用数值归约的新方法，比如对数值流求和的 sum ，找到最大元素的max。此外有在必要时再把它们转换回对象流的方法。要记住的是，这些特化的原因并不在于流的复杂性，而是箱造成的复杂性——即类似 int 和 Integer 之间的效率差异。

1.映射到数值流

将流转换为特化版本的常用方法是 mapToInt 、 mapToDouble 和 mapToLong 。这些方法和前面说的 map 方法的工作方式一样，只是它们返回的是一个特化流，而不是 Stream<T> 。例如，我可以像下面这样用 mapToInt 对 menu 中的卡路里求和：

```
int calories = menu.stream()
        // 返回一个IntStream
        .mapToInt(Dish::getCalories)
        .sum();
```

这里， mapToInt 会从每道菜中提取热量（用一个 Integer 表示），并返回一个 IntStream （而不是一个 Stream<Integer> ）。然后你就可以调用 IntStream 接口中定义的 sum 方法，对卡路里求和了！请注意，如果流是空的， sum 默认返回 0 。 IntStream 还支持其他的方便方法，如 max 、 min 、 average 等。

2.转换回对象流

同样，一旦有了数值流，你可能会想把它转换回非特化流。例如， IntStream 上的操作只能产生原始整数： IntStream 的 map 操作接受的Lambda必须接受 int 并返回 int （一个 IntUnaryOperator ）。但是你可能想要生成另一类值，比如 Dish 。为此，你需要访问 Stream 接口中定义的那些更广义的操作。要把原始流转换成一般流（每个 int 都会装箱成一个 Integer ），可以使用 boxed 方法，如下所示：

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> stream = intStream.boxed();
```

3.默认值 OptionalInt

求和的那个例子很容易，因为它有一个默认值： 0 。但是，如果你要计算 IntStream 中的最大元素，就得换个法子了，因为 0 是错误的结果。如何区分没有元素的流和最大值真的是 0 的流呢？前面我们介绍了 Optional 类，这是一个可以表示值存在或不存在的容器。 Optional 可以用 Integer 、 String 等参考类型来参数化。对于三种原始流特化，也分别有一个 Optional 原始类型特化版本： OptionalInt 、 OptionalDouble 和 OptionalLong 。

例如，要找到 IntStream 中的最大元素，可以调用 max 方法，它会返回一个 OptionalInt ：

```
OptionalInt maxCalories = menu.stream()
                .mapToInt(Dish::getCalories)
                .max();
```

现在，如果没有最大值的话，你就可以显式处理 OptionalInt 去定义一个默认值了：

```
int max = maxCalories.orElse(1);
```

数值范围

和数字打交道时，有一个常用的东西就是数值范围。比如，假设你想要生成1和100之间的所有数字。Java 8引入了两个可以用于 IntStream 和 LongStream 的静态方法，帮助生成这种范围：range 和 rangeClosed 。这两个方法都是第一个参数接受起始值，第二个参数接受结束值。但 range 是不包含结束值的，而 rangeClosed 则包含结束值。让我们来看一个例子：

```
// 一个从1到100的偶数流 包含结束值
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
        .filter(n -> n % 2 == 0);
// 从1到100共有50个偶数
System.out.println(evenNumbers.count());
```

这里我们用了 rangeClosed 方法来生成1到100之间的所有数字。它会产生一个流，然后你可以链接 filter 方法，只选出偶数。到目前为止还没有进行任何计算。最后，你对生成的流调用 count 。因为 count 是一个终端操作，所以它会处理流，并返回结果 50 ，这正是1到100（包括两端）中所有偶数的个数。请注意，比较一下，如果改用 IntStream.range(1, 100) ，则结果将会是 49 个偶数，因为 range 是不包含结束值的。

构建流

希望到现在，我们已经让你相信，流对于表达数据处理查询是非常强大而有用的。到目前为止，你已经能够使用 stream 方法从集合生成流了。此外，我们还介绍了如何根据数值范围创建数值流。但创建流的方法还有许多！本节将介绍如何从值序列、数组、文件来创建流，甚至由生成函来创建无限流！

由值创建流

你可以使用静态方法 Stream.of ，通过显式值创建一个流。它可以接受任意数量的参数。例如，以下代码直接使用 Stream.of 创建了一个字符串流。然后，你可以将字符串转换为大写，再一个个打印出来：

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

你可以使用 empty 得到一个空流，如下所示：

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

由数组创建流

我们可以使用静态方法 Arrays.stream 从数组创建一个流。它接受一个数组作为参数。例如，

我们可以将一个原始类型 int 的数组转换成一个 IntStream ，如下所示：

```
int[] numbers = {2, 3, 5, 7, 11, 13};
// 总和41
int sum = Arrays.stream(numbers).sum();
```

由文件生成流

Java中用于处理文件等I/O操作的NIO API（非阻塞 I/O）已更新，以便利用Stream API。

java.nio.file.Files 中的很多静态方法都会返回一个流。例如，一个很有用的方法是

Files.lines ，它会返回一个由指定文件中的各行构成的字符串流。使用我们迄今所学的内容，我们可用这个方法看看一个文件中有多少各不相同的词：

```
long uniqueWords;
try (Stream<String> lines = Files.lines(Paths.get(ClassLoader.getSystemResource("data.txt").t
URI()),
        Charset.defaultCharset())) {
    uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))
            .distinct()
            .count();
    System.out.println("uniqueWords:" + uniqueWords);
} catch (IOException e) {
    e.fillInStackTrace();
} catch (URISyntaxException e) {
    e.printStackTrace();
}
```

你可以使用 Files.lines 得到一个流，其中的每个元素都是给定文件中的一行。然后，你

可以对 line 调用 split 方法将行拆分成单词。应该注意的是，你该如何使用 flatMap 产生一个扁平的词流，而不是给每一行生成一个单词流。最后，把 distinct 和 count 方法链接起来，数数流中有多各不相同的单词。

由函数生成流：创建无限流

Stream API提供了两个静态方法来从函数生成流： Stream.iterate 和 Stream.generate 。

这两个操作可以创建所谓的无限流：不像从固定集合创建的流那样有固定大小的流。由 iterate和 generate 产生的流会用给定的函数按需创建值，因此可以无穷无尽地计算下去！一般来说，应该使用 limit n) 来对这种流加以限制，以避免打印无穷多个值。

1.迭代

我们先来看一个 iterate 的简单例子，然后再解释：

```
Stream.iterate(0, n -> n + 2)
        .limit(10)
        .forEach(System.out::println);
```

iterate 方法接受一个初始值（在这里是 0 ），还有一个依次应用在每个产生的新值上的

Lambda （ UnaryOperator<t> 类型）。这里，我们使用Lambda n -> n + 2 ，返回的是前一个元加上2。因此，iterate方法生成了一个所有正偶数的流：流的第一个元素是初始值 0 。然后加上 2 来

成新的值 2 ，再加上 2 来得到新的值 4 ，以此类推。这种 iterate 操作基本上是顺序的，因为结果取于前一次应用。请注意，此操作将生成一个无限流——这个流没有结尾，因为值是按需计算的，可以远计算下去。我们说这个流是无界的。正如我们前面所讨论的，这是流和集合之间的一个关键区别。们使用limit方法来显式限制流的大小。这里只选择了前10个偶数。然后可以调用 forEach 终端操作消费流，并分别打印每个元素。

2.生成

与 iterate 方法类似， generate 方法也可让你按需生成一个无限流。但 generate 不是依次对每个新生成的值应用函数的。它接受一个 Supplier<T> 类型的Lambda提供新的值。我们先来看一个简单的用法：

```
Stream.generate(Math::random)
            .limit(5)
            .forEach(System.out::println);
```

这段代码将生成一个流，其中有五个0到1之间的随机双精度数。例如，运行一次得到了下面的结果：

```
0.8404010101858976
0.03607897810804739
0.025199243727344833
0.8368092999566692
0.14685668895309267
```

Math.Random 静态方法被用作新值生成器。同样，你可以用 limit 方法显式限制流的大小，否则流将会无限长。

你可能想知道， generate 方法还有什么用途。我们使用的供应源（指向 Math.random 的方法引用）是无状态的：它不会在任何地方记录任何值，以备以后计算使用。但供应源不一定是无状态。你可以创建存储状态的供应源，它可以修改状态，并在为流生成下一个值时使用。

我们在这个例子中会使用 IntStream 说明避免装箱操作的代码。 IntStream 的 generate 方法会接受一个 IntSupplier ，而不是 Supplier<t> 。例如，可以这样来生成一个全是1的无限流：

```
IntStream ones = IntStream.generate(() -> 1);
```

还记得第三章的笔记中，Lambda允许你创建函数式接口的实例，只要直接内联提供方法的实现就可以。你也可以像下面这样，通过实现 IntSupplier 接口中定义的 getAsInt 方法显式传递一个象（虽然这看起来是无缘无故地绕圈子，也请你耐心看）：

```
IntStream twos = IntStream.generate(new IntSupplier(){
        @Override
        public int getAsInt(){
            return 2;
        }
    });
```

generate 方法将使用给定的供应源，并反复调用 getAsInt 方法，而这个方法总是返回 2 。但这里使用的匿名类和Lambda的区别在于，匿名类可以通过字段定义状态，而状态又可以用 getAsInt 方法来修改。这是一个副作用的例子。我们迄今见过的所有Lambda都是没有副作用的；它

没有改变任何状态。

总结

这一章的东西很多，收获也很多！现在你可以更高效地处理集合了。事实上，流让你可以简洁地表达杂的数据处理查询。此外，流可以透明地并行化。以下是我们应从本章中学到的关键概念。

这一章的读书笔记中，我们学习和了解到了：

1. Streams API可以表达复杂的数据处理查询。
2. 你可以使用 filter 、 distinct 、 skip 和 limit 对流做筛选和切片。
3. 你可以使用 map 和 flatMap 提取或转换流中的元素。
4. 你可以使用 findFirst 和 findAny 方法查找流中的元素。你可以用 allMatch、noneMatch 和 any atch 方法让流匹配给定的谓词。
5. 这些方法都利用了短路：找到结果就立即停止计算；没有必要处理整个流。
6. 你可以利用 reduce 方法将流中所有的元素迭代合并成一个结果，例如求和或查找最大元素。
7. filter 和 map 等操作是无状态的，它们并不存储任何状态。 reduce 等操作要存储状态才能计算出一个值。 sorted 和 distinct 等操作也要存储状态，因为它们需要把流中的所有元素缓存起来才能返回一个新的流。这种操作称为有状态操作。
8. 流有三种基本的原始类型特化： IntStream 、 DoubleStream 和 LongStream 。它们的操作也有相应的特化。
9. 流不仅可以从集合创建，也可从值、数组、文件以及 iterate 与 generate 等特定方法创建。
10. 无限流是没有固定大小的流。

代码

Github： chap5

Gitee： chap5