链滴

《Java8 实战》- 第六章读书笔记（用流收集数据 -01）

作者：Not-Found
原文链接：https://ld246.com/article/1537678195252
来源网站：链滴

用流收集数据

我们在前一章中学到，流可以用类似于数据库的操作帮助你处理集合。你可以把Java 8的流看作花哨懒惰的数据集迭代器。它们支持两种类型的操作：中间操作（如 filter 或 map ）和终端操作（如 cout 、 findFirst 、 forEach 和 reduce ）。中间操作可以链接起来，将一个流转换为另一个流。这些操不会消耗流，其目的是建立一个流水线。与此相反，终端操作会消耗流，以产生一个最终结果，例如回流中的最大元素。它们通常可以通过优化流水线来缩短计算时间。

我们已经在前面用过了 collect 终端操作了，当时主要是用来把 Stream 中所有的元素结合成一个 List 。在本章中，你会发现 collect 是一个归约操作，就像 reduce 一样可以接受各种做法作为参数，将中的元素累积成一个汇总结果。具体的做法是通过定义新的Collector 接口来定义的，因此区分 Collection 、 Collector 和 collect 是很重要的。

现在，我们来看一个例子，看看我们用collect和收集器能做什么。

1. 对一个交易列表按照货币分组，获得该货币所有的交易总额和（返回一个 Map<Currency,Integer> ）。

2. 将交易列表分成两组：贵的和不贵的（返回一个 Map<Boolean, List <Transaction>> ）。

3. 创建多级分组，比如按城市对交易分组，然后进一步按照贵或不贵分组（返回一个 Map<Boolean, List<Transaction>> ）。

我们首先来看一个利用收集器的例子，想象一下，你有一个Transaction构成的List，并且想按照名义币进行分组。在没有Lambda的Java里，哪怕像这种简单的用例实现起来都很啰嗦，就像下面这样：

```
// 建立累积交易分组的Map
Map<Currency, List<Transaction>> transactionsByCurrencies = new HashMap<>(16);
// 迭代 Transaction 的 List
for (Transaction transaction : transactions) {
    // 提取 Transaction的货币
    Currency currency = transaction.getCurrency();
    List<Transaction> transactionsForCurrency = transactionsByCurrencies.get(currency);
    // 如果分组 Map 中没有这种货币的条目，就创建一个
    if (transactionsForCurrency == null) {
        transactionsForCurrency = new ArrayList<>();
        transactionsByCurrencies.put(currency, transactionsForCurrency);
    }
    // 将当前遍历的 Transaction加入同一货币的 Transaction 的 List
    transactionsForCurrency.add(transaction);
}
System.out.println(transactionsByCurrencies);
```

如果你是一位经验丰富的Java程序员，写这种东西可能挺顺手的，不过你必须承认，做这么简单的一事就得写很多代码。更糟糕的是，读起来比写起来更费劲！代码的目的并不容易看出来，尽管换作白的话是很直截了当的：“把列表中的交易按货币分组。”你在本章中会学到，用Stream 中 collect 法的一个更通用的 Collector 参数，你就可以用一句话实现完全相同的结果，而用不着使用上一章那个 toList 的特殊情况了：

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream().collect(groupingBy(Transaction::getCurrency));
```

这一比差得还真多，对吧？

收集器简介

前一个例子清楚地展示了函数式编程相对于指令式编程的一个主要优势：你只需指出希望的结果——做什么”，而不用操心执行的步骤——“如何做”。在上一个例子里，传递给 collect方法的参数是 Collector 接口的一个实现，也就是给 Stream 中元素做汇总的方法。上一章里的toList 只是说“按顺序每个元素生成一个列表”；在本例中， groupingBy 说的是“生成一个Map，它的键是（货币）桶值则是桶中那些元素的列表”。要是做多级分组，指令式和函数式之间的区别就会更加明显：由于需好多层嵌套循环和条件，指令式代码很快就变得更难阅读、更难维护、更难修改。

收集器用作高级归约

刚刚的结论又引出了优秀的函数式API设计的另一个好处：更易复合和重用。收集器非常有用，因为它可以简洁而灵活地定义collect用来生成结果集合的标准。更具体地说，对流调用collect方法将对流的元素触发一个归约操作（由Collector来参数化）。一般来说， Collector 会对元素应用一个转换函（很多时候是不体现任何效果的恒等转换，例如 toList ），并将结果累积在一个数据结构中，从而产这一过程的最终输出。例如，在前面所示的交易分组的例子中，转换函数提取了每笔交易的货币，随使用货币作为键，将交易本身累积在生成的 Map 中。

归约和汇总

为了说明从 Collectors 工厂类中能创建出多少种收集器实例，我们重用一下前一章的例子：包含一张肴列表的菜单！就像你刚刚看到的，在需要将流项目重组成集合时，一般会使用收集器（ Stream 方法collect的参数）。再宽泛一点来说，但凡要把流中所有的项目合并成一个结果时就可以用。这个结果以是任何类型，可以复杂如代表一棵树的多级映射，或是简单如一个整数——也许代表了菜单的热量和。

我们先来举一个简单的例子，利用 counting 工厂方法返回的收集器，数一数菜单里有多少

种菜：

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

这还可以写得更为直接：

```
long howManyDishes = menu.stream().count();
```

counting 收集器在和其他收集器联合使用的时候特别有用，后面会谈到这一点。

查找流中的最大值和最小值

假设你想要找出菜单中热量最高的菜。你可以使用两个收集器， Collectors.maxBy和Collectors.miny ，来计算流中的最大或最小值。这两个收集器接收一个 Comparator 参数来

比较流中的元素。你可以创建一个 Comparator来根据所含热量对菜肴进行比较，并把它传递给

Collectors.maxBy：

```
List<Dish> menu =  Dish.MENU;
Comparator<Dish> dishCaloriesComparator =
        Comparator.comparingInt(Dish::getCalories);
Optional<Dish> mostCalorieDish =
        menu.stream().max(dishCaloriesComparator);
System.out.println(mostCalorieDish.get());
```

你可能在想 Optional<Dish> 是怎么回事。要回答这个问题，我们需要问“要是 menu 为空怎么办。那就没有要返回的菜了！Java 8引入了 Optional ，它是一个容器，可以包含也可以不包含值。这它完美地代表了可能也可能不返回菜肴的情况。

另一个常见的返回单个值的归约操作是对流中对象的一个数值字段求和。或者你可能想要求平均数。种操作被称为汇总操作。让我们来看看如何使用收集器来表达汇总操作。

汇总

Collectors 类专门为汇总提供了一个工厂方法： Collectors.summingInt 。它可接受一个把对象映为求和所需 int 的函数，并返回一个收集器；该收集器在传递给普通的 collect 方法后即执行我们需的汇总操作。举个例子来说，你可以这样求出菜单列表的总热量：

```
List<Dish> menu =  Dish.MENU;
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

除了Collectors.summingInt，还有Collectors.summingLong 和Collectors.summingDouble 方法作用完全一样，可以用于求和字段为 long 或 double 的情况。

但汇总不仅仅是求和；还有 Collectors.averagingInt ，连同对应的 averagingLong 和 averagingDouble 可以计算数值的平均数：

```
List<Dish> menu =  Dish.MENU;
double avgCalories =
            menu.stream().collect(averagingInt(Dish::getCalories));
```

到目前为止，你已经看到了如何使用收集器来给流中的元素计数，找到这些元素数值属性的最大值和小值，以及计算其总和和平均值。不过很多时候，你可能想要得到两个或更多这样的结果，而且你希只需一次操作就可以完成。在这种情况下，你可以使用 summarizingInt 工厂方法返回的收集器。例，通过一次 summarizing 操作你可以就数出菜单中元素的个数，并得到菜肴热量总和、平均值、最值和最小值：

```
List<Dish> menu =  Dish.MENU;
IntSummaryStatistics menuStatistics =
      menu.stream().collect(summarizingInt(Dish::getCalories));
System.out.println(menuStatistics.getMax());
System.out.println(menuStatistics.getAverage());
System.out.println(menuStatistics.getMin());
System.out.println(menuStatistics.getCount());
System.out.println(menuStatistics.getSum());
```

同样，相应的 summarizingLong 和 summarizingDouble 工厂方法有相关的LongSummaryStatistis 和 DoubleSummaryStatistics 类型，适用于收集的属性是原始类型 long 或double 的情况。

连接字符串

joining 工厂方法返回的收集器会把对流中每一个对象应用 toString 方法得到的所有字符串连接成一个字符串。这意味着你把菜单中所有菜肴的名称连接起来，如下所示：

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

请注意， joining 在内部使用了 StringBuilder 来把生成的字符串逐个追加起来。结果：

```
porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon
```

但该字符串的可读性并不好。幸好， joining 工厂方法有一个重载版本可以接受元素之间的分界符，这样你就可以得到一个逗号分隔的菜肴名称列表：

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

结果：

```
pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon
```

到目前为止，我们已经探讨了各种将流归约到一个值的收集器。在下一节中，我们会展示为什么所有种形式的归约过程，其实都是 Collectors.reducing 工厂方法提供的更广义归约收集器的特殊情况。

广义的归约汇总

事实上，我们已经讨论的所有收集器，都是一个可以用 reducing 工厂方法定义的归约过程的特殊情而已。 Collectors.reducing 工厂方法是所有这些特殊情况的一般化。可以说，先前讨论的案例仅仅为了方便程序员而已。（但是，请记得方便程序员和可读性是头等大事！）例如，可以用 reducing 法创建的收集器来计算你菜单的总热量，如下所示：

```
List<Dish> menu =  Dish.MENU;
int totalCalories = menu.stream().collect(reducing(
        0, Dish::getCalories, (i, j) -> i + j));
System.out.println(totalCalories);
```

它需要三个参数：

1. 第一个参数是归约操作的起始值，也是流中没有元素时的返回值，所以很显然对于数值和而言0是个合适的值。
2. 第二个参数是Lambda的语法糖，将菜肴转换成一个表示其所含热量的 int 。
3. 第三个参数是一个 BinaryOperator ，将两个项目累积成一个同类型的值。这里它就是对两个 int 求和。

同样，你可以使用下面这样单参数形式的 reducing 来找到热量最高的菜，如下所示：

```
Optional<Dish> mostCalorieDish =
        menu.stream().collect(reducing(
            (d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2));
```

你可以把单参数 reducing 工厂方法创建的收集器看作三参数方法的特殊情况，它把流中的第一个项作为起点，把恒等函数（即一个函数仅仅是返回其输入参数）作为一个转换函数。

收集框架的灵活性：以不同的方法执行同样的操作

你还可以进一步简化前面使用 reducing 收集器的求和例子——引用 Integer 类的 sum 方法，而不去写一个表达同一操作的Lambda表达式。这会得到以下程序：

```
int totalCalories2 = menu.stream()
                .collect(reducing(0, // 初始值
                        Dish::getCalories, // 转换函数
                        Integer::sum)); // 积累函数
```

使用语法糖，能帮助我们简化一部分代码。

还有另外一种方法不使用收集器也能执行相同操作——将菜肴流映射为每一道菜的热量，然后用前一版本中使用的方法引用来归约得到的流：

```
int totalCalories =
        menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
```

请注意，就像流的任何单参数 reduce 操作一样， reduce(Integer::sum) 返回的不是 int而是 Optional Integer> ，以便在空流的情况下安全地执行归约操作。然后你只需用 Optional对象中的 get 方法来取里面的值就行了。请注意，在这种情况下使用 get 方法是安全的，只是因为你已经确定菜肴流不为 。一般来说，使用允许提供默认值的方法，如 orElse 或 orElseGet来解开Optional中包含的值更为全。最后，更简洁的方法是把流映射到一个 IntStream ，然后调用 sum 方法，你也可以得到相同的果：

```
int totalCalories = menu.stream().mapToInt(Dish::getCalories).sum();
```

根据情况选择最佳解决方案

这再次说明了，函数式编程（特别是Java 8的 Collections 框架中加入的基于函数式风格原理设计的API）通常提供了多种方法来执行同一个操作。这个例子还说明，收集器在某种程度上比Stream 接口直接提供的方法用起来更复杂，但好处在于它们能提供更高水平的抽象和概括，也更容易重用和自定 。在《Java8实战》中的的建议是，尽可能为手头的问题探索不同的解决方案，但在通用的方案里面始终选择最专门化的一个。无论是从可读性还是性能上看，这一般都是最好的决定。例如，要计菜单总热量，我们更倾向于最后一个解决方案（使用 IntStream ），因为它最简明，也很可能最易读。同 ，它也是性能最好的一个，因为 IntStream 可以让我们避免自动拆箱操作，也就是从Integer到int的式转换，它在这里毫无用处。

分组

一个常见的数据库操作是根据一个或多个属性对集合中的项目进行分组。就像前面讲到按货币对交易行分组的例子一样，如果用指令式风格来实现的话，这个操作可能会很麻烦、啰嗦而且容易出错。但 ，如果用Java 8所推崇的函数式风格来重写的话，就很容易转化为一个非常容易看懂的语句。我们来看这个功能的第二个例子：假设你要把菜单中的菜按照类型进行分类，有肉的放一组，有鱼的放一组其他的都放另一组。用 Collectors.groupingBy 工厂方法返回的收集器就可以轻松地完成这项任务，下所示：

```
Map<Dish.Type, List<Dish>> dishesByType =
                    menu.stream().collect(groupingBy(Dish::getType));
```

其结果是下面的 Map：

```
{OTHER=[Dish{name='french fries'}, Dish{name='rice'}, Dish{name='season fruit'}, Dish{name=
pizza'}], MEAT=[Dish{name='pork'}, Dish{name='beef'}, Dish{name='chicken'}], FISH=[Dish{n
me='prawns'}, Dish{name='salmon'}]}
```

这里，你给 groupingBy 方法传递了一个 Function （以方法引用的形式），它提取了流中每一道 Dis 的 Dish.Type 。我们把这个 Function 叫作分类函数，因为它用来把流中的元素分成不同的组。分组作的结果是一个 Map ，把分组函数返回的值作为映射的键，把流中所有具有这个分类值的项目的列作为对应的映射值。在菜单分类的例子中，键就是菜的类型，值就是包含所有对应类型的菜肴的列表。

但是，分类函数不一定像方法引用那样可用，因为你想用以分类的条件可能比简单的属性访问器要复 。例如，你可能想把热量不到400卡路里的菜划分为“低热量” （diet），热量400到700卡路里的菜为“普通” （normal），高于700卡路里的划为“高热量” （fat）。由于 Dish 类的作者没有把这个

作写成一个方法，你无法使用方法引用，但你可以把这个逻辑写成Lambda表达式：

```
public enum CaloricLevel {
    /**
     * 卡路里等级
     */
    DIET, NORMAL, FAT
}

 Map<Dish.CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream().collect(
            groupingBy(dish -> {
                if (dish.getCalories() <= 400) {
                    return Dish.CaloricLevel.DIET;
                } else if (dish.getCalories() <= 700) {
                    return Dish.CaloricLevel.NORMAL;
                } else {
                    return Dish.CaloricLevel.FAT;
                }
            }));
```

多级分组

要实现多级分组，我们可以使用一个由双参数版本的 Collectors.groupingBy 工厂方法创建的收集器它除了普通的分类函数之外，还可以接受 collector 类型的第二个参数。那么要进行二级分组的话，们可以把一个内层 groupingBy 传递给外层groupingBy ，并定义一个为流中项目分类的二级标准。

```
Map<Dish.Type, Map<Dish.CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
            menu.stream().collect(
                    groupingBy(Dish::getType,
                            groupingBy(dish -> {
                                if (dish.getCalories() <= 400) {
                                    return Dish.CaloricLevel.DIET;
                                } else if (dish.getCalories() <= 700) {
                                    return Dish.CaloricLevel.NORMAL;
                                } else {
                                    return Dish.CaloricLevel.FAT;
                                }
                            })
                    )
            );
```

这个二级分组的结果就是像下面这样的两级 Map ：

```
{OTHER={DIET=[Dish{name='rice'}, Dish{name='season fruit'}], NORMAL=[Dish{name='french fries'}, Dish{name='pizza'}]}, MEAT={DIET=[Dish{name='chicken'}], FAT=[Dish{name='pork'}], ORMAL=[Dish{name='beef'}]}, FISH={DIET=[Dish{name='prawns'}], NORMAL=[Dish{name='salmon'}]}}
```

这里的外层 Map 的键就是第一级分类函数生成的值： “fish, meat, other” ，而这个 Map 的值又是个 Map ，键是二级分类函数生成的值： “normal, diet, fat” 。最后，第二级 map 的值是流中元素成的 List ，是分别应用第一级和第二级分类函数所得到的对应第一级和第二级键的值： “salmon、pi za…” 这种多级分组操作可以扩展至任意层级，n级分组就会得到一个代表n级树形结构的n级Map 。

一般来说，把 groupingBy 看作“桶”比较容易明白。第一个 groupingBy 给每个键建立了一个桶。

后再用下游的收集器去收集每个桶中的元素，以此得到n级分组。

按子组收集数据

在上一节中，我们看到可以把第二个 groupingBy 收集器传递给外层收集器来实现多级分组。但进一说，传递给第一个 groupingBy 的第二个收集器可以是任何类型，而不一定是另一个 groupingBy 。如，要数一数菜单中每类菜有多少个，可以传递 counting 收集器作为groupingBy 收集器的第二个数：

```
 Map<Dish.Type, Long> typesCount = menu.stream().collect(groupingBy(Dish::getType, counting()));
```

其结果是下面的 Map ：

```
{OTHER=4, MEAT=3, FISH=2}
```

还要注意，普通的单参数 groupingBy(f) （其中 f 是分类函数）实际上是 groupingBy(f,toList()) 的便写法。

再举一个例子，你可以把前面用于查找菜单中热量最高的菜肴的收集器改一改，按照菜的类型分类：

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
        menu.stream()
            .collect(groupingBy(Dish::getType,
                maxBy(comparingInt(Dish::getCalories))));
```

这个分组的结果显然是一个 map ，以 Dish 的类型作为键，以包装了该类型中热量最高的 Dish的 Optional<Dish> 作为值：

```
{OTHER=Optional[Dish{name='pizza'}], MEAT=Optional[Dish{name='pork'}], FISH=Optional[Dish{name='salmon'}]}
```

把收集器的结果转换为另一种类型

因为分组操作的 Map 结果中的每个值上包装的 Optional 没什么用，所以你可能想要把它们去掉。做到这一点，或者更一般地来说，把收集器返回的结果转换为另一种类型，你可以使用Collectors.collctingAndThen 工厂方法返回的收集器，如下所示。

查找每个子组中热量最高的 Dish：

```
List<Dish> menu = Dish.MENU;
    Map<Dish.Type, Dish> mostCaloricByType =
        menu.stream()
            .collect(groupingBy(Dish::getType, // 分类函数
                collectingAndThen(
                    maxBy(comparingInt(Dish::getCalories)), // 包装后的收集器
                    Optional::get))); // 转换函数
```

这个工厂方法接受两个参数——要转换的收集器以及转换函数，并返回另一个收集器。这个收集器相于旧收集器的一个包装， collect 操作的最后一步就是将返回值用转换函数做一个映射。在这里，被起来的收集器就是用 maxBy 建立的那个，而转换函数 Optional::get 则把返回的 Optional 中的值提出来。前面已经说过，这个操作放在这里是安全的，因为 reducing收集器永远都不会返回 Optional.mpty() 。其结果是下面的 Map ：

```
{OTHER=Dish{name='pizza'}, MEAT=Dish{name='pork'}, FISH=Dish{name='salmon'}}
```

把好几个收集器嵌套起来很常见，它们之间到底发生了什么可能不那么明显。从最外层开始逐层向里注意以下几点：

1. 收集器用虚线表示，因此 groupingBy 是最外层，根据菜肴的类型把菜单流分组，得到三个子流。

2. groupingBy 收集器包裹着 collectingAndThen 收集器，因此分组操作得到的每个子流都用这第个收集器做进一步归约。

3. collectingAndThen 收集器又包裹着第三个收集器 maxBy 。

4. 随后由归约收集器进行子流的归约操作，然后包含它的 collectingAndThen 收集器会对其结果应用Optional:get 转换函数。

5. 对三个子流分别执行这一过程并转换而得到的三个值，也就是各个类型中热量最高的Dish ，将成为groupingBy 收集器返回的 Map 中与各个分类键（ Dish 的类型）相关联的值。

与 groupingBy 联合使用的其他收集器的例子

一般来说，通过 groupingBy 工厂方法的第二个参数传递的收集器将会对分到同一组中的所有流元素行进一步归约操作。例如，你还重用求出所有菜肴热量总和的收集器，不过这次是对每一组 Dish 求：

```
Map<Dish.Type, Integer> totalCaloriesByType = menu.stream()
        .collect(groupingBy(Dish::getType,
            summingInt(Dish::getCalories)));
```

然而常常和 groupingBy 联合使用的另一个收集器是 mapping 方法生成的。这个方法接受两个参数一个函数对流中的元素做变换，另一个则将变换的结果对象收集起来。其目的是在累加之前对每个输元素应用一个映射函数，这样就可以让接受特定类型元素的收集器适应不同类型的对象。我们来看一使用这个收集器的实际例子。比方说你想要知道，对于每种类型的 Dish ，菜单中都有哪些 CaloricLeel 。我们可以把 groupingBy 和 mapping 收集器结合起来，如下所示：

```
Map<Dish.Type, Set<Dish.CaloricLevel>> caloricLevelsByType =
        menu.stream().collect(
            groupingBy(Dish::getType, mapping(
                dish -> {
                    if (dish.getCalories() <= 400) {
                        return Dish.CaloricLevel.DIET;
                    } else if (dish.getCalories() <= 700) {
                        return Dish.CaloricLevel.NORMAL;
                    } else {
                        return Dish.CaloricLevel.FAT;
                    }
                },
                toSet())));
```

传递给映射方法的转换函数将 Dish 映射成了它的CaloricLevel ： 生成的CaloricLevel 流传递给一个 tSet 收集器，它和 toList 类似，不过是把流中的元素累积到一个 Set 而不是 List 中，以便仅保留各不同的值。如先前的示例所示，这个映射收集器将会收集分组函数生成的各个子流中的元素，让你得到样的 Map 结果：

```
{OTHER=[DIET, NORMAL], MEAT=[DIET, FAT, NORMAL], FISH=[DIET, NORMAL]}
```

由此你就可以轻松地做出选择了。如果你想吃鱼并且在减肥，那很容易找到一道菜；同样，如果你饥辘辘，想要很多热量的话，菜单中肉类部分就可以满足你的饕餮之欲了。请注意在上一个示例中，对返回的 Set 是什么类型并没有任何保证。但通过使用 toCollection ，你就可以有更多的控制。例如你可以给它传递一个构造函数引用来要求 HashSet ：

```
Map<Dish.Type, Set<Dish.CaloricLevel>> caloricLevelsByType =
        menu.stream().collect(
                groupingBy(Dish::getType, mapping(
                        dish -> {
                            if (dish.getCalories() <= 400) {
                                return Dish.CaloricLevel.DIET;
                            } else if (dish.getCalories() <= 700) {
                                return Dish.CaloricLevel.NORMAL;
                            } else {
                                return Dish.CaloricLevel.FAT;
                            }
                        },
                        toCollection(HashSet::new))));
```

使用流收集数据这一章，内容是比较多的，使用分组等特性能帮助我们简化很大一部分的工作，从而高我们的开发效率。

代码

Github:chap6

Gitee:chap6