链滴

《Java8 实战》- 第六章读书笔记（用流收集数据 -02）

作者：Not-Found
原文链接：https://ld246.com/article/1538888596907
来源网站：链滴

使用流收集数据

分区

分区是分组的特殊情况：由一个谓词（返回一个布尔值的函数）作为分类函数，它称分区函数。分区数返回一个布尔值，这意味着得到的分组 Map 的键类型是 Boolean ，于是它最多可以分为两组——rue 是一组， false 是一组。例如，如果你是素食者或是请了一位素食的朋友来共进晚餐，可能会想把菜单按照素食和非素食分开：

```
Map<Boolean, List<Dish>> partitionedMenu =
            // 分区函数
            menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

这会返回下面的 Map ：

```
{false=[Dish{name='pork'}, Dish{name='beef'}, Dish{name='chicken'}, Dish{name='prawns'}, D
sh{name='salmon'}],
true=[Dish{name='french fries'}, Dish{name='rice'}, Dish{name='season fruit'}, Dish{name='piz
a'}]}
```

那么通过 Map 中键为 true 的值，就可以找出所有的素食菜肴了：

```
List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

请注意，用同样的分区谓词，对菜单 List 创建的流作筛选，然后把结果收集到另外一个 List中也可以得相同的结果：

```
List<Dish> vegetarianDishes =
                menu.stream().filter(Dish::isVegetarian).collect(toList());
```

分区的优势

分区的好处在于保留了分区函数返回 true 或 false 的两套流元素列表。在上一个例子中，要得到非素 Dish 的 List ，你可以使用两个筛选操作来访问 partitionedMenu 这个 Map 中 false键的值：一个用谓词，一个利用该谓词的非。而且就像你在分组中看到的， partitioningBy工厂方法有一个重载版，可以像下面这样传递第二个收集器：

```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
            menu.stream().collect(
                // 分区函数
                partitioningBy(Dish::isVegetarian,
                    // 第二个收集器
                    groupingBy(Dish::getType)));
```

这将产生一个二级 Map ：

```
{false={MEAT=[Dish{name='pork'}, Dish{name='beef'}, Dish{name='chicken'}], FISH=[Dish{na
e='prawns'}, Dish{name='salmon'}]},
true={OTHER=[Dish{name='french fries'}, Dish{name='rice'}, Dish{name='season fruit'}, Dish{
ame='pizza'}]}}
```

这里，对于分区产生的素食和非素食子流，分别按类型对菜肴分组，得到了一个二级 Map，和上面

类似。再举一个例子，你可以重用前面的代码来找到素食和非素食中热量最高的菜：

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian = menu.stream().collect(
            partitioningBy(Dish::isVegetarian, collectingAndThen(
                    maxBy(comparingInt(Dish::getCalories)),
                    Optional::get
            )));
```

这将产生以下结果：

```
{false=Dish{name='pork'}, true=Dish{name='pizza'}}
```

你可以把分区看作分组一种特殊情况。 groupingBy 和partitioningBy 收集器之间的相似之处并不止此。

将数字按质数和非质数分区

假设你要写一个方法，它接受参数 int n，并将前n个自然数分为质数和非质数。但首先，找出能够测某一个待测数字是否是质数的谓词会很有帮助：

```
private static boolean isPrime(int candidate) {
    // 产生一个自然数范围，从2开始，直至但不包括待测数
    return IntStream.range(2, candidate)
            // 如果待测数字不能被流中任何数字整除则返回 true
            .noneMatch(i -> candidate % i == 0);
}
```

一个简单的优化是仅测试小于等于待测数平方根的因子：

```
private static boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
            .noneMatch(i -> candidate % i == 0);
}
```

现在最主要的一部分工作已经做好了。为了把前n个数字分为质数和非质数，只要创建一个包含这n个的流，用刚刚写的 isPrime 方法作为谓词，再给 partitioningBy 收集器归约就好了：

```
private static Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
            .collect(
                    partitioningBy(candidate -> isPrime(candidate)));
}
```

现在我们已经讨论过了 Collectors 类的静态工厂方法能够创建的所有收集器，并介绍了使用它们的实例子。

收集器接口

Collector 接口包含了一系列方法，为实现具体的归约操作（即收集器）提供了范本。我们已经看过了 Collector 接口中实现的许多收集器，例如 toList 或 groupingBy 。这也意味着，你可以为 Collector 接口提供自己的实现，从而自由地创建自定义归约操作。

要开始使用 Collector 接口，我们先看看本章开始时讲到的一个收集器—— toList 工厂方法，它会把

中的所有元素收集成一个 List 。我们当时说在日常工作中经常会用到这个收集器，而且它也是写起来较直观的一个，至少理论上如此。通过仔细研究这个收集器是怎么实现的，我们可以很好地了解 Collector 接口是怎么定义的，以及它的方法所返回的函数在内部是如何为collect 方法所用的。

首先让我们在下面的列表中看看 Collector 接口的定义，它列出了接口的签名以及声明的五个方法。

```
public interface Collector<T, A, R> {
      Supplier<A> supplier();
      BiConsumer<A, T> accumulator();
      Function<A, R> finisher();
      BinaryOperator<A> combiner();
      Set<Characteristics> characteristics();
}
```

本列表适用以下定义。

1. T 是流中要收集的项目的泛型。
2. A 是累加器的类型，累加器是在收集过程中用于累积部分结果的对象。
3. R 是收集操作得到的对象（通常但并不一定是集合）的类型。

例如，你可以实现一个 ToListCollector<T> 类，将 Stream<T> 中的所有元素收集到一个List<T> ，它的签名如下：

```
public class ToListCollector<T> implements Collector<T, List<T>, List<T>>
```

我们很快就会澄清，这里用于累积的对象也将是收集过程的最终结果。

理解 Collector 接口声明的方法

现在我们可以一个个来分析 Collector 接口声明的五个方法了。通过分析，你会注意到，前四个方法会返回一个会被 collect 方法调用的函数，而第五个方法 characteristics 则提供了一系列特征，也就一个提示列表，告诉 collect 方法在执行归约操作的时候可以应用哪些优化（比如并行化）。

1. 建立新的结果容器： supplier 方法

supplier 方法必须返回一个结果为空的 Supplier ，也就是一个无参数函数，在调用时它会创建一个的累加器实例，供数据收集过程使用。很明显，对于将累加器本身作为结果返回的收集器，比如我们的 ToListCollector ，在对空流执行操作的时候，这个空的累加器也代表了收集过程的结果。在我们的 ToListCollector 中， supplier 返回一个空的 List ，如下所示：

```
@Override
public Supplier<List<T>> supplier() {
    return () -> new ArrayList<>();
}
```

请注意你也可以只传递一个构造函数引用：

```
@Override
public Supplier<List<T>> supplier() {
    return ArrayList::new;
}
```

2. 将元素添加到结果容器： accumulator 方法

accumulator 方法会返回执行归约操作的函数。当遍历到流中第n个元素时，这个函数执行时会有两参数：保存归约结果的累加器（已收集了流中的前 n-1 个项目），还有第n个元素本身。该函数将返 void ，因为累加器是原位更新，即函数的执行改变了它的内部状态以体现遍历的元素的效果。对于ToLstCollector ，这个函数仅仅会把当前项目添加至已经遍历过的项目的列表：

```
@Override
public BiConsumer<List<T>, T> accumulator() {
    return (list, item) -> list.add(item);
}
```

你也可以使用方法引用，这会更为简洁：

```
@Override
public BiConsumer<List<T>, T> accumulator() {
    return List::add;
}
```

3. 对结果容器应用最终转换： finisher 方法

在遍历完流后， finisher 方法必须返回在累积过程的最后要调用的一个函数，以便将累加器对象转换整个集合操作的最终结果。通常，就像 ToListCollector 的情况一样，累加器对象恰好符合预期的最结果，因此无需进行转换。所以 finisher 方法只需返回 identity 函数：

```
@Override
public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}
```

这三个方法已经足以对流进行循序规约。实践中的实现细节可能还要复杂一点，一方面是应为流的延性质，可能在collect操作之前还需完成其他中间操作的流水线，另一方面则是理论上可能要进行并行约。

4. 合并两个结果容器： combiner 方法

四个方法中的最后一个————————combiner方法会返回一个供归约操作的使用函数，它定义了对流的个子部分进行并行处理时，各个子部分归约所得的累加器要如何合并。对于toList而言，这个方法的现非常简单，只要把从流的第二个部分收集到的项目列表加到遍历第一部分时得到的列表后面就行了：

```
@Override
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1;
    };
}
```

有了这第四个方法，就可以对流进行并行归约了。它会用到Java7中引入的分支/合并框架和Spliterato抽象。

5. characteristics 方法

最后一个方法—— characteristics 会返回一个不可变的 Characteristics 集合，它定义了收集器的行——尤其是关于流是否可以并行归约，以及可以使用哪些优化的提示。Characteristics 是一个包含个项目的枚举。

1. UNORDERED ——归约结果不受流中项目的遍历和累积顺序的影响。

2. CONCURRENT —— accumulator 函数可以从多个线程同时调用，且该收集器可以并行归约流。果收集器没有标为 UNORDERED ，那它仅在用于无序数据源时才可以并行归约。

3. IDENTITY_FINISH ——这表明完成器方法返回的函数是一个恒等函数，可以跳过。这种情况下，加器对象将会直接用作归约过程的最终结果。这也意味着，将累加器 A 不加检查地转换为结果 R 是全的。

我们迄今开发的 ToListCollector 是 IDENTITY_FINISH 的，因为用来累积流中元素的List 已经是我要的最终结果，用不着进一步转换了，但它并不是 UNORDERED ，因为用在有序流上的时候，我们是希望顺序能够保留在得到的 List 中。最后，它是 CONCURRENT 的，但我们刚才说过了，仅仅在后的数据源无序时才会并行处理。

全部融合到一起

前一小节中谈到的五个方法足够我们开发自己的 ToListCollector 了。你可以把它们都融合起来，如面的代码清单所示。

```
public class ToListCollector<T> implements Collector<T, List<T>, List<T>> {
    @Override
    public Supplier<List<T>> supplier() {
        return ArrayList::new;
    }

    @Override
    public BiConsumer<List<T>, T> accumulator() {
        return List::add;
    }

    @Override
    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);
            return list1;
        };
    }

    @Override
    public Function<List<T>, List<T>> finisher() {
        return Function.identity();
    }

    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(Characteristics.IDENTITY_FINISH, Charact
ristics.CONCURRENT));
    }
}
```

请注意，这个是实现与Collections.toList()方法并不完全相同，但区别仅仅是一些小的优化。这些优的一个主要方面是Java API所提供的收集器在需要返回空列表时使用了 Collections.emptyList() 这个例（singleton）。这意味着它可安全地替代原生Java，来收集菜单流中的所有 Dish 的列表：

```
List<Dish> dishes = menuStream.collect(new ToListCollector<>());
```

这个实现和标准的

```
List<Dish> dishes = menuStream.collect(toList());
```

构造之间的其他差异在于 toList 是一个工厂，而 ToListCollector 必须用 new 来实例化。

进行自定义收集而不去实现 Collector

对于 IDENTITY_FINISH 的收集操作，还有一种方法可以得到同样的结果而无需从头实现新的 Collecto s 接口。 Stream 有一个重载的 collect 方法可以接受另外三个函数—— supplier 、accumulator 和 ombiner ，其语义和 Collector 接口的相应方法返回的函数完全相同。所以比如说，我们可以像下面样把菜肴流中的项目收集到一个 List 中：

```
List<Dish> dishes = menuStream.collect(
            ArrayList::new,
            List::add,
            List::addAll);
```

我们认为，这第二种形式虽然比前一个写法更为紧凑和简洁，却不那么易读。此外，以恰当的类来实自己的自定义收集器有助于重用并可避免代码重复。另外值得注意的是，这第二个collect 方法不能递任何 Characteristics ，所以它永远都是一个 IDENTITY_FINISH 和CONCURRENT 但并非 UNORD RED 的收集器。

在下一节中，我们一起来实现一个收集器的，让我们对收集器的新知识更上一层楼。你将会为一个更复杂，但更为具体、更有说服力的用例开发自己的自定义收集器。

开发你自己的收集器以获得更好的性能

我们用 Collectors 类提供的一个方便的工厂方法创建了一个收集器，它将前n个自然数划分为质数和质数，如下所示。

将前n个自然数按质数和非质数分区:

```
private static Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
            .collect(
                    partitioningBy(candidate -> isPrime(candidate)));
}
```

当时，通过限制除数不超过被测试数的平方根，我们对最初的 isPrime 方法做了一些改进：

```
private static boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
            .noneMatch(i -> candidate % i == 0);
}
```

还有没有办法来获得更好的性能呢？答案是“有”，但为此你必须开发一个自定义收集器。

仅用质数做除数

一个可能的优化是仅仅看看被测试数是不是能够被质数整除。要是除数本身都不是质数就用不着测了所以我们可以仅仅用被测试数之前的质数来测试。然而我们目前所见的预定义收集器的问题，也就是须自己开发一个收集器的原因在于，在收集过程中是没有办法访问部分结果的。这意味着，当测试某

个数字是否是质数的时候，你没法访问目前已经找到的其他质数的列表。

假设你有这个列表，那就可以把它传给 isPrime 方法，将方法重写如下：

```
private static boolean isPrime(List<Integer> primes, int candidate) {
    return primes.stream().noneMatch(i -> candidate % i == 0);
}
```

而且还应该应用先前的优化，仅仅用小于被测数平方根的质数来测试。因此，你需要想办法在下一个数大于被测数平方根时立即停止测试。不幸的是，Stream API中没有这样一种方法。你可以使用 filter p -> p <= candidateRoot) 来筛选出小于被测数平方根的质数。但 filter要处理整个流才能返回恰当结果。如果质数和非质数的列表都非常大，这就是个问题了。你用不着这样做；你只需在质数大于被数平方根的时候停下来就可以了。因此，我们会创建一个名为 takeWhile 的方法，给定一个排序列表一个谓词，它会返回元素满足谓词的最长前缀：

```
public static <A> List<A> takeWhile(List<A> list, Predicate<A> p) {
    int i = 0;
    for (A item : list) {
        if (!p.test(item)) {
            return list.subList(0, i);
        }
        i++;
    }
    return list;
}
```

利用这个方法，你就可以优化 isPrime 方法，只用不大于被测数平方根的质数去测试了：

```
private static boolean isPrime(List<Integer> primes, int candidate){
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return takeWhile(primes, i -> i <= candidateRoot)
            .stream()
            .noneMatch(p -> candidate % p == 0);
}
```

请注意，这个 takeWhile 实现是即时的。理想情况下，我们会想要一个延迟求值的takeWhile ，这就可以和 noneMatch 操作合并。不幸的是，这样的实现超出了本章的范围，你需要了解Stream API 实现才行。

有了这个新的 isPrime 方法在手，你就可以实现自己的自定义收集器了。首先要声明一个实现 Collector 接口的新类，然后要开发 Collector 接口所需的五个方法。

1. 第一步：定义 Collector 类的签名

让我们从类签名开始吧，记得 Collector 接口的定义是：

```
public interface Collector<T, A, R>
```

其中 T 、 A 和 R 分别是流中元素的类型、用于累积部分结果的对象类型，以及 collect 操作最终结的类型。这里应该收集 Integer 流，而累加器和结果类型则都是 Map<Boolean,List<Integer>>，是 true 和 false ，值则分别是质数和非质数的 List ：

```
public class PrimeNumbersCollector implements Collector<Integer, Map<Boolean, List<Integ
r>>,
        Map<Boolean, List<Integer>>>
```

2. 第二步：实现归约过程

接下来，你需要实现 Collector 接口中声明的五个方法。 supplier 方法会返回一个在调用时创建累加的函数：

```
@Override
public Supplier<Map<Boolean, List<Integer>>> supplier() {
    return () -> new HashMap<Boolean, List<Integer>>(2) {
        {
            put(true, new ArrayList<>());
            put(false, new ArrayList<>());
        }
    };
}
```

这里不但创建了累积器的Map，还为true和false两个键下面出实话了对应的空列表。在收集过程中会质数和非指数分别添加到这里。收集器重要的方法是accumulator，因为它定义了如何收集流中元素逻辑。这里它也是实现了前面所讲的优化的关键。现在在任何一次迭代中，都可以访问收集过程的部结果，也就是包含迄今找到的质数的累加器：

```
@Override
public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
    return ((Map<Boolean, List<Integer>> acc, Integer candidate) -> acc.get(isPrime(acc.get(tre), candidate)).add(candidate));
}
```

在这个个方法中，你调用了isPrime方法，将待测试是否为质数的数以及迄今为止找到的质数列表（就是累积Map中true键对应的值）传递给它。这次调用的结果随后被用作获取质数或非质数列表的键这样就可以把新的被测数添加到恰当的列表中。

3.第三步：让收集器并行工作（如果可能）

下一个方法要在并行收集时把两个部分累加器合并起来，这里，它只需要合并两个Map，即将第二个ap中质数和非质数列表中的所有数字合并到第一个Map的对应列表中就行了：

```
@Override
public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
    return (Map<Boolean, List<Integer>> map1, Map<Boolean, List<Integer>> map2) -> {
        map1.get(true).addAll(map2.get(true));
        map1.get(false).addAll(map2.get(false));
        return map1;
    };
}
```

请注意，实际上这个收集器是不能并行的，因为该算法本身是顺序的。这意味着永远都不会调用combner方法，你可以把它的实现留空。为了让这个例子完整，我们还是决定实现它。

4.第四步：finisher方法和收集器的characteristics方法

最后两个方法实现都很简单。前面说过，accumulator正好就是收集器的结果，也用不着进一步转换那么finisher方法就返回identity函数：

```
@Override
public Function<Map<Boolean, List<Integer>>, Map<Boolean, List<Integer>>> finisher() {
    return Function.identity();
```

```
}
```

就characteristics方法而言，我们已经说过，它既不是CONCURRENT也不是UNOREDERED，但却是DENTITY_FINISH的：

```
@Override
public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(Characteristics.IDENTITY_FINISH));
}
```

现在，你可以用这个新的自定义收集器来替代partitioningBy工厂方法创建的那个，并获得完全相同结果了：

```
private static Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector(int n) {
    return IntStream.rangeClosed(2, n).boxed()
            .collect(new PrimeNumbersCollector());
}
Map<Boolean, List<Integer>> primes = partitionPrimesWithCustomCollector(10);
// {false=[4, 6, 8, 9, 10], true=[2, 3, 5, 7]}
System.out.println(primes);
```

收集器性能比较

用partitioningBy工厂方法穿件的收集器和你刚刚开发的自定义收集器在功能上是一样的，但是我们有实现用自定义收集器超越partitioningBy收集器性能的目标呢？现在让我们写个小程序测试一下吧：

```
public class CollectorHarness {
    public static void main(String[] args) {
        long fastest = Long.MAX_VALUE;
        // 运行十次
        for (int i = 0; i < 10; i++) {
            long start = System.nanoTime();
            // 将前100万个自然数按指数和非质数区分
            partitionPrimes(1_000_000);
            long duration = (System.nanoTime() - start) / 1_000_000;
            // 检查这个执行是否是最快的一个
            if (duration < fastest) {
                fastest = duration;
            }
            System.out.println("done in " + duration);
        }
        System.out.println("Fastest execution done in " + fastest + " msecs");
    }
}
```

在因特尔I5 6200U 2.4HGz的笔记上运行得到以下的结果：

```
done in 976
done in 1091
done in 866
done in 867
done in 760
done in 759
done in 777
```

```
done in 894
done in 765
done in 763
Fastest execution done in 759 msecs
```

现在把测试框架的 partitionPrimes 换成 partitionPrimesWithCustomCollector，以便测试我们开的自定义收集器的性能。

```
public class CollectorHarness {
    public static void main(String[] args) {
        excute(PrimeNumbersCollectorExample::partitionPrimesWithCustomCollector);
    }

    private static void excute(Consumer<Integer> primePartitioner) {
        long fastest = Long.MAX_VALUE;
        // 运行十次
        for (int i = 0; i < 10; i++) {
            long start = System.nanoTime();
            // 将前100万个自然数按指数和非质数区分
            // partitionPrimes(1_000_000);
            primePartitioner.accept(1_000_000);
            long duration = (System.nanoTime() - start) / 1_000_000;
            // 检查这个执行是否是最快的一个
            if (duration < fastest) {
                fastest = duration;
            }
            System.out.println("done in " + duration);
        }
        System.out.println("Fastest execution done in " + fastest + " msecs");
    }
}
```

现在，程序打印：

```
done in 703
done in 649
done in 715
done in 434
done in 386
done in 403
done in 449
done in 416
done in 353
done in 405
Fastest execution done in 353 msecs
```

还不错！看来我们没有白费功夫开发这个自定义收集器。

总结

1. collect 是一个终端操作，它接受的参数是将流中元素累积到汇总结果的各种方式（称为收集器）。
2. 预定义收集器包括将流元素归约和汇总到一个值，例如计算最小值、最大值或平均值。
3. 预定义收集器可以用 groupingBy 对流中元素进行分组，或用 partitioningBy 进行分区。

4. 收集器可以高效地复合起来，进行多级分组、分区和归约。
5. 你可以实现 Collector 接口中定义的方法来开发你自己的收集器。

代码

Github:chap6

Gitee:chap6