# Part 2 第二部分

# 函数式数据处理

本书第二部分深入探索了新的Stream API——它可以让你编写功能强大的代码，用声明性的方式处理数据集。学完第二部分，你将充分理解流是什么，以及如何在代码中使用它来简明而高效地处理数据集。

第4章介绍了流的概念，并解释了它与集合的异同。

第5章详细讨论了表达复杂数据处理查询可以使用的流操作。我们会谈到很多模式，如筛选、切片、查找、匹配、映射和归约。

第6章介绍了收集器——Stream API的一个功能，可以让你表达更为复杂的数据处理查询。

在第7章中，你将了解流为何可以自动并行执行，并利用多核架构的优势。此外，你还会了解到要避免的若干陷阱，以便正确而高效地使用并行流。

第 4 章

# 引入流

4

本章内容

- 什么是流
- 集合与流
- 内部迭代与外部迭代
- 中间操作与终端操作

集合是Java中使用最多的API。要是没有集合，还能做什么呢？几乎每个Java应用程序都会制造和处理集合。集合对于很多编程任务来说都是非常基本的：它们可以让你把数据分组并加以处理。为了解释集合是怎么工作的，想象一下你准备列出一系列菜，组成一张菜单，然后再遍历一遍，把每盘菜的热量加起来。你可能想选出那些热量比较低的菜，组成一张健康的特殊菜单。尽管集合对于几乎任何一个Java应用都是不可或缺的，但集合操作却远远算不上完美。

- 很多业务逻辑都涉及类似于数据库的操作，比如对几道菜按照类别进行分组（比如全素菜肴），或查找出最贵的菜。你自己用迭代器重新实现过这些操作多少遍？大部分数据库都允许你声明式地指定这些操作。比如，以下SQL查询语句就可以选出热量较低的菜肴名称：`SELECT name FROM dishes WHERE calorie < 400`。你看，你不需要实现如何根据菜肴的属性进行筛选（比如利用迭代器和累加器），你只需要表达你想要什么。这个基本的思路意味着，你用不着担心怎么去显式地实现这些查询语句——都替你办好了！怎么到了集合这里就不能这样了呢？
- 要是要处理大量元素又该怎么办呢？为了提高性能，你需要并行处理，并利用多核架构。但写并行代码比用迭代器还要复杂，而且调试起来也够受的！

那Java语言的设计者能做些什么，来帮助你节约宝贵的时间，让你这个程序员活得轻松一点儿呢？你可能已经猜到了，答案就是流。

## 4.1 流是什么

流是Java API的新成员，它允许你以声明性方式处理数据集合（通过查询语句来表达，而不是临时编写一个实现）。就现在来说，你可以把它们看成遍历数据集的高级迭代器。此外，流还

可以透明地并行处理，你无需写任何多线程代码了！我们会在第7章中详细解释流和并行化是怎么工作的。我们简单看看使用流的好处吧。下面两段代码都是用来返回低热量的菜肴名称的，并按照卡路里排序，一个是用Java 7写的，另一个是用Java 8的流写的。比较一下。不用太担心Java 8代码怎么写，我们在接下来的几节里会详细解释。

之前（Java 7）：

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.getCalories() < 400){
        lowCaloricDishes.add(d);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(), d2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes){
    lowCaloricDishesName.add(d.getName());
}
```

用累加器筛选元素

用匿名类对菜肴排序

处理排序后的菜名列表

在这段代码中，你用了一个“垃圾变量”lowCaloricDishes。它唯一的作用就是作为一次性的中间容器。在Java 8中，实现的细节被放在它本该归属的库里了。

之后（Java 8）：

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
                menu.stream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dish::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

选出400卡路里以下的菜肴

按照卡路里排序

提取菜肴的名称

将所有名称保存在List中

为了利用多核架构并行执行这段代码，你只需要把stream()换成parallelStream()：

```
List<String> lowCaloricDishesName =
                menu.parallelStream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dishes::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

你可能会想，在调用parallelStream方法的时候到底发生了什么。用了多少个线程？对性能有多大提升？第7章会详细讨论这些问题。现在，你可以看出，从软件工程师的角度来看，新的方法有几个显而易见的好处。

- 代码是以声明性方式写的：说明想要完成什么（筛选热量低的菜肴）而不是说明如何实

现一个操作（利用循环和if条件等控制流语句）。你在前面的章节中也看到了，这种方法加上行为参数化让你可以轻松应对变化的需求：你很容易再创建一个代码版本，利用Lambda表达式来筛选高卡路里的菜肴，而用不着去复制粘贴代码。

- 你可以把几个基础操作链接起来，来表达复杂的数据处理流水线（在filter后面接上sorted、map和collect操作，如图4-1所示），同时保持代码清晰可读。filter的结果被传给了sorted方法，再传给map方法，最后传给collect方法。

因为filter、sorted、map和collect等操作是与具体线程模型无关的高层次构件，所以它们的内部实现可以是单线程的，也可能透明地充分利用你的多核架构！在实践中，这意味着你用不着为了让某些数据处理任务并行而去操心线程和锁了，Stream API都替你做好了！

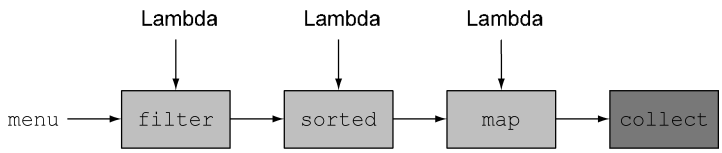


图4-1　将流操作链接起来构成流的流水线

新的Stream API表达能力非常强。比如在读完本章以及第5章、第6章之后，你就可以写出像下面这样的代码：

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

我们在第6章中解释这个例子。简单来说就是，按照Map里面的类别对菜肴进行分组。比如，Map可能包含下列结果：

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

想想要是改用循环这种典型的指令型编程方式该怎么实现吧。别浪费太多时间了。拥抱这一章和接下来几章中强大的流吧！

**其他库：Guava、Apache和lambdaj**

为了给Java程序员提供更好的库操作集合，前人已经做过了很多尝试。比如，Guava就是谷歌创建的一个很流行的库。它提供了multimaps和multisets等额外的容器类。Apache Commons Collections库也提供了类似的功能。最后，本书作者Mario Fusco编写的lambdaj受到函数式编程的启发，也提供了很多声明性操作集合的工具。

如今Java 8自带了官方库，可以以更加声明性的方式操作集合了。

总结一下，Java 8中的Stream API可以让你写出这样的代码：

- 声明性——更简洁，更易读

- 可复合——更灵活
- 可并行——性能更好

在本章剩下的部分和下一章中，我们会使用这样一个例子：一个menu，它只是一张菜肴列表。

```
List<Dish> menu = Arrays.asList(
    new Dish("pork", false, 800, Dish.Type.MEAT),
    new Dish("beef", false, 700, Dish.Type.MEAT),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("season fruit", true, 120, Dish.Type.OTHER),
    new Dish("pizza", true, 550, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("salmon", false, 450, Dish.Type.FISH) );
```

Dish类的定义是：

```
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public Dish(String name, boolean vegetarian, int calories, Type type) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public int getCalories() {
        return calories;
    }

    public Type getType() {
        return type;
    }

    @Override
    public String toString() {
        return name;
    }
```

```
    public enum Type { MEAT, FISH, OTHER }
}
```

现在就来仔细探讨一下怎么使用Stream API。我们会用流与集合做类比，做点儿铺垫。下一章会详细讨论可以用来表达复杂数据处理查询的流操作。我们会谈到很多模式，如筛选、切片、查找、匹配、映射和归约，还会提供很多测验和练习来加深你的理解。

接下来，我们会讨论如何创建和操纵数字流，比如生成一个偶数流，或是勾股数流。最后，我们会讨论如何从不同的源（比如文件）创建流。还会讨论如何生成一个具有无穷多元素的流——这用集合肯定是搞不定了！

## 4.2　流简介

要讨论流，我们先来谈谈集合，这是最容易上手的方式了。Java 8中的集合支持一个新的stream方法，它会返回一个流（接口定义在java.util.stream.Stream里）。你在后面会看到，还有很多其他的方法可以得到流，比如利用数值范围或从I/O资源生成流元素。

那么，*流*到底是什么呢？简短的定义就是“从支持数据处理操作的源生成的元素序列”。让我们一步步剖析这个定义。

- *元素序列*——就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。因为集合是数据结构，所以它的主要目的是以特定的时间/空间复杂度存储和访问元素（如ArrayList 与 LinkedList）。但流的目的在于表达计算，比如你前面见到的filter、sorted和map。集合讲的是数据，流讲的是计算。我们会在后面几节中详细解释这个思想。
- *源*——流会使用一个提供数据的源，如集合、数组或输入/输出资源。请注意，从有序集合生成流时会保留原有的顺序。由列表生成的流，其元素顺序与列表一致。
- *数据处理操作*——流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match、sort等。流操作可以顺序执行，也可并行执行。

此外，流操作有两个重要的特点。

- *流水线*——很多流操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的流水线。这让我们下一章中的一些优化成为可能，如*延迟*和*短路*。流水线的操作可以看作对数据源进行数据库式查询。
- *内部迭代*——与使用迭代器显式迭代的集合不同，流的迭代操作是在背后进行的。我们在第1章中简要地提到了这个思想，下一节会再谈到它。

让我们来看一段能够体现所有这些概念的代码：

```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
  menu.stream()
      .filter(d -> d.getCalories() > 300)
```

从menu获得流（菜肴列表）

建立操作流水线：首先选出高热量的菜肴