函数式编程与流

John MacCormick，迪金森学院，2023年2月。

摘要：本文旨在作为一章独立的教材，用于在主要教学语言为 Java 的课程中介绍一些函数式编程概念。有关此方法的详细信息，请参阅 J. MacCormick 撰写的论文《函数式编程：在数据结构课程中使用 Java》，该论文发表于《大学计算科学期刊》（2023年）。

1 函数式编程的一些背景

编程语言的设计与实现是计算机科学中一个广泛而重要的分支。在本章中，我们将探讨函数式编程，这是程序设计语言理论中最重要的思想之一。编程语言通常被描述为属于不同的类别或范式。这些程序设计语言范式包括命令式 语言、函数式 语言以及逻辑编程语言。大多数现代语言都包含多种范式的特性。例如，Java 和 Python 主要是作为命令式语言设计的，但也包含许多函数式编程的特点。Lisp、F# 和 Haskell 则是专为函数式语言而设计的例子。

本章首先探讨函数式编程中最基础的思想之一：*Lambda* 表达式。接着，我们将了解如何在 Java Stream API 中使用 Lambda 表达式，高效而优雅地处理数据集。

1.1 关于本章中 Java 和 Python 用法的说明

尽管 Java 和 Python 并不是特别适合作为函数式编程语言的示例，本章中的示例仍只使用 Java 和 Python。研究一种更加纯粹的函数式语言会让我们偏离主题太远。如果某些 Java 示例看起来有些别扭，请记住，我们有意在理解函数式编程思想与利用自己对 Java 的熟悉程度之间寻求折中。如果你不熟悉 Python，也不必担心。与 Java 相比，用 Python 展示某些函数式编程思想更加容易。因此，我们会先介绍一些 Python 示例。不过，即使不深入理解 Python 示例，也足以理解 Java 示例。

2 Lambda 表达式

在本章余下部分中，函数一词通常指编程语言中的子程序，它可以接收参数并返回值。不同的编程语言对函数的称呼各不相同。例如，在 Java 中，函数称为方法。在 Python 中，函数就称为函数。

函数式编程语言的一个关键特点是，它们将函数视为与其他数据类型相同的对象。这通常用“函数是一等公民”或“函数是一等对象”这样的表述来描述。函数具备一等地位所带来的最重要影响是，在函数式语言中，一个函数可以作为另一个函数的参数。

2.1 Python 中作为参数的函数

在 Python 中尤其容易演示这一点：

```
def add5(x):
    return x + 5

def multBy3IfPositive(x):
    if x > 0:
        return 3 * x
    否则：
        返回 0

def applyToSeven(f):
    返回 f(7)

def applyToMinusNine(f):
    返回 f(-9)
```

这里，函数 `applyToSeven` 和 `applyToMinusNine` 都接受一个参数，该参数应当是一个函数。例如，我们可以将 `add5` 作为 `applyToSeven`的参数：

```
>>> applyToSeven(add5) 12
```

当然，如果将不同的函数作为参数传入，我们就能得到不同的结果：

```
>>> applyToSeven(multBy3IfPositive) 21
```

示例题 1。

Che通过计算下面两个函数调用的结果来检查你对本节内容的理解 :

```
>>> applyToMinusNine(add5)
```

```
>>> applyToMinusNine(multBy3IfPositive)
```

所有示例题的解答都列在本章末尾。

2.2 Java 中作为参数的函数

与 Python 相比，在 Java 中将函数作为参数传递并不那么容易。下面的 Java 代码是上述 Python 代码最简单的等价实现。正如你将看到的，它相对而言比较冗长。它依靠 Java 接口来模拟将函数作为参数传递的能力。在此示例中，我们导入 `Function<T, R>` 接口，该接口来自 `java.util.function`，这是支持函数式编程的 Java 包。 `Function<T, R>` 接口表示一个接受单个 `T` 类型参数并返回 `R` 类型值的函数。由于 `add5` 函数接受一个整数参数并返回一个整数，我们可以通过创建一个实现 `add5` 函数效果的 `Add5` 类，使其实现 `Function<Integer, Integer>` 接口。在下面的 `main()` 方法中，我们创建了 `Add5` 类的一个实例。该实例的名称是 `add5`，并且

我们可以将这个实例（它确实是一个 Java 对象）作为参数传递给 `applyToSeven()` 和 `applyToMinusNine()` 方法。

```
导入 java.util.function.Function；公共类 FunctionParameterDemo { 公共静态类 Add5 实现 Function<Integer, Integer> { 公共 Integer apply(Integer x) { 返回 x + 5; } } 公共静态类 MultBy3IfPositive 实现 Function<Integer, Integer> { 公共 Integer apply(Integer x) { 如果 (x > 0) { 返回 3 * x; } 否则 { 返回 0; } } } 公共静态 Integer applyToSeven(Function<Integer, Integer> f) { 返回 f.apply(7); } 公共静态 Integer applyToMinusNine(Function<Integer, Integer> f) { 返回 f.apply(-9); } 公共静态 void main(String[] args) { Add5 add5 = new Add5(); MultBy3IfPositive multBy3IfPositive = new MultBy3IfPositive(); int val
                    1 =                    5          1 = 12        applyToSeven(add); //
                    2 =                    3
val int val applyToSeven(multByIfPositive); // val
                                                                  2 = 21        } }
```

示例问题 2。

确定以下两个方法调用的值，假设它们被插入到上面的 `main()` 方法末尾：

```
applyToMinusNine(add5)
applyToMinusNine(multBy3IfPositive)
```

2.3 Java 中的函数式接口和 Lambda 表达式

请注意，`add5` 和 `multBy3IfPositive` 所实现的接口具有以下特殊属性：它只有一个抽象方法。该方法名为 `apply()`。在 Java 中，恰好只有一个抽象方法的接口称为 函数式接口。稍后我们将看到，函数式接口非常重要，因为可以使用*Lambda* 表达式高效而优雅地实现它们。Lambda 表达式是函数式接口的简写形式，可以让我们避免使用上述 Java 函数式编程中那种笨拙的写法。为了了解 Lambda 表达式的实际应用，我们先回到 Python。

2.4 命名值与匿名值

在任何编程语言编写的计算机程序中，有些值通常是命名的，而另一些值则是匿名的。例如，下面是在 Python 中打印 5 的平方根的两种不同方法：

```
print(math.sqrt(5)) # 匿名

x = 5print(math.sqrt(x)) # 命名
```

在第一次调用 `sqrt()`时，值 5 是匿名的，因为没有使用变量名来引用它。在第二次调用 `sqrt()`时，则使用名称 `x`来引用值 5。

2.5 Python 中的 lambda 关键字

回想一下，在函数式编程中，我们可以将函数作为参数。因此，传递函数作为参数时，应该可以采用同样的两种方法：既可以使用具名函数，也可以使用匿名函数。我们已经看过如何使用具名函数来实现这一点。下面再次列出前面提到的一个例子，以便具体说明：

```
>>> applyToSeven(add5)
```

这里将具名函数 `add5` 作为参数传递给函数 `applyToSeven()` 的调用。但如何使用匿名函数来实现这一点呢？在 Python 中，我们可以使用关键字 `lambda` 创建匿名函数，如下例所示。

```
>>> applyToSeven(lambda x: x+5)
```

因此，Lambda 表达式只是一种描述函数而不为其命名的方式。Python 代码片段“`lambda x: x+5`”表示“接受参数 `x` 并返回 `x+5` 的函数”。如果将 `lambda` 称为其他名称，例如 `anonFunction`，或许会更容易理解，但采用关键字 `lambda` 有充分的历史和理论依据。“Lambda”是希腊字母 λ 的名称。在 20 世纪 30 年代，数学家阿隆佐·丘奇使用希腊字母 λ 作为描述使用函数进行计算的框架的主要符号——这一框架如今称为 *Lambda* 演算。Lambda 演算是函数式编程和计算理论的核心，但要在此研究这种联系就会离题太远。

总结一下：在 Python 中，Lambda 表达式定义了一个匿名函数。我们完全不必对 Lambda 表达式感到困惑。如果你看到包含令人困惑的 Lambda 表达式的代码，只需先创建一个能够实现该 Lambda 表达式效果的具名函数，然后用新名称替换 Lambda 表达式即可。例如，Python 代码片段

```
执行奇怪的操作("无意义", \       lambda apple, banana, grape: apple +
banana.gimble(2*grape))
```

可以改写为

```
def weirdFunction(apple, banana, grape): return
apple +  banana.gimble(2*grape)

performStrangeAction("nonsense", weirdFunction)
```

上面的示例还展示了 Python 中的 Lambda 表达式如何描述具有多个参数的函数。

示例问题 3。

假设　前面给出的定义，以下 Python 代码片段的输出是什么　?

```
applyToSeven(lambda potato: potato%4  +  potato*potato)
applyToMinusNine(lambda oak: math.factorial(oak+12) * oak)
```

2.6 Java 中的 Lambda 表达式

在 Java 中，Lambda 表达式不使用关键字 `lambda`，而是在函数参数后立即使用箭头表示法“`->`”。例如，Python 代码片段“`lambda x: x+5`”在 Java 中的等价写法是“`x -> x+5`”。在这两种情况下，Lambda 表达式的含义大致都是“接收参数 `x` 并返回 `x+5` 的函数”。Java Lambda 表达式的真正含义更加复杂，但这里不作详细说明。只需知道，Java Lambda 表达式使用前面提到的函数式接口，创建匿名类和匿名实例，从而实现所需的函数式编程效果。

我们可以使用 Lambda 表达式重写前面的 Java 示例。`applyToSeven()` 和 `applyToMinusNine()` 的声明保持不变，这里为方便起见再次列出：

```
    public static Integer applyToSeven(Function<Integer, Integer>  f) { return f.apply(7); }
public static Integer applyToMinusNine(Function<Integer, Integer>  f) { return f.apply(-9); }
```

其余代码紧凑得多，因为无需声明任何实现函数式接口的类。我们的主方法可以改为

```
public static void main(String[] args) {
        整数 val3 = applyToSeven(x -> x + 5); // 值3 = 12
        int val4 = applyToSeven(x -> {
                if (x > 0) {
                        return 3 * x;
                } else {
                        return 0;
                }
        }); // val4 = 21
}
```

请注意，这里使用了跨越多行并包含多个代码块的 Lambda 表达式。实际应用中，Lambda 表达式通常简短而简单，但也可以根据需要变得任意复杂。

示例问题 4。

假设将以下两个方法调用插入到上面的 `main()` 方法末尾，求它们的值：

```
int val5 = applyToMinusNine(x -> (x + 1) * (x + 2)); int val6 =
applyToMinusNine(z -> { if (z > 10) { return 100; } else if (z < -100) { return -100; }
else { return z * 10; } });
```

Java 中的 Lambda 表达式也可以包含多个参数。这可以通过在“`->`”符号前的括号内列出参数来实现。例如，Lambda 表达式 `(x, y) -> x*x +2*y` 可以表示如下所示的方法 `f()`。

```
public int f(int x, int y) {
        return x*x + 2*y;
}
```

示例题 5。

将数学函数 $g(u, v, w) = \sqrt{u^2 + v^2 + w^2}$ 表示为 Java Lambda 表达式。

3 Java 流 API

在 Java 中，流<T> 是一个用于对 T 类型的对象序列执行操作的接口。此外，还有用于对某些基本类型的序列执行操作的专用流：IntStream 用于 int 值，LongStream 用于 long 值，以及 DoubleStream 用于 double 值。

常用于 流的操作包括 count()、filter()过滤操作、map()、mapToInt()、foreach()和 reduce()。 IntStream、LongStream和 DoubleStream还有 sum()求和操作。我们只学习这七种操作。Java API 文档还列出了其他可用操作。

每种操作都被定义为中间操作 或终端操作。我们将学习的七种常见操作分类如下。

- 中间操作： filter()过滤操作、map()、mapToInt()。
- 终端操作： count()、foreach()、reduce()、sum()求和操作。

要对 流执行计算，可以依次应用零个或多个中间操作，最后再应用一个终端操作。例如，计算可能由以下一系列操作构成： filter()过滤操作、filter()过滤操作、map()、filter()过滤操作、count()。

3.1 创建流

在 Java 中，创建流的方法有很多。下面的代码片段展示了其中几种方法。

```
// 方法 1：直接通过 Stream.of() 从数组创建 String[] 数组 = { "bat", "cat", "bird", "
mad", "catch", "ditch" }; Stream<String> stream1 = Stream.of(array); // 方法 2：直
接通过 Stream.of() 从多个参数创建 Stream<String> stream2 = Stream.of("bat", "
cat", "bird", "mad", "catch", "ditch"); // 方法 3：使用 Java 集合的 stream() 方法转换任
意 Java 集合 List<String> list = Arrays.asList("bat", "cat", "bird", "mad", "catch", "
ditch"); Stream<String> stream3 = list.stream(); // 方法 4：使用 Files.lines 从文件创
建流 Stream<String> stream4 = Files.lines(Paths.get("data/words.txt")); // 方法 5：
使用 range() 或 rangeClosed() IntStream range1 = IntStream.range(5, 10); //
5,6,7,8,9 IntStream range2 = IntStream.rangeClosed(5, 10); // 5,6,7,8,9,10
```

示例题 6。

(i) 编写代码，创建一个流，其中包含以下 Double对象序列：23.4、69.7、-25.88、31.3363。

（ii）重复第（i）部分，这次创建一个包含基本 Double 值的流 .

（iii）编写代码，创建一个由 100 到 200（含）之间的整数组成的流。

下一节将介绍七种常用操作，同时逐步加深我们对如何将这些操作组合起来，以构成更有趣的流计算的理解。

3.2 流操作

现在，我们将详细考察七种流操作。

3.2.1 count()

流的count()操作返回流中的元素数量。

```
流<字符串> 流 = Stream.of("bat", "cat", "bird"); long numElements =
stream.count(); // numElements = 3
```

示例题 7。

编写一段使用 Stream API 统计名为"GreatGatsby.txt"的文件行数的代码。

3.2.2 sum()

sum()求和操作会返回流中元素的总和。它只能用于专门的数值流 IntStream、LongStream 和 DoubleStream，如下例所示。

```
DoubleStream 流 = DoubleStream.of(1.5, 2.4, -0.1); double total =
stream.sum(); // total = 3.8
```

示例问题8。

编写一段代码，使用 Stream API 将从 27 到 159（含）的整数相加。提示：IntStream.range() 或 IntStream.rangeClosed()可以轻松实现这一点。

3.2.3 filter()过滤操作

filter()过滤操作会对输入流中的每个元素应用布尔测试函数。未通过测试（即返回 false）的元素会被丢弃，而通过测试（即返回 true）的元素则会进入输出流，由计算中的下一个操作处理。这是我们首次在 流中使用 Lambda 表达式，因为布尔测试函数可以用 Lambda 表达式描述。请看下面的例子。

```
流<字符串> 流 = Stream.of("bat", "cat", "bird", "mad", "catch", "ditch"); 流<字符串>
newStream = 流.filter(word -> word.startsWith("ca"));
```

这里，流是计算的输入。它是一个流<字符串>，包含元素 "bat"、"cat"、"bird"、"mad"、"catch" 和 "ditch"。计算的输出是newStream，其数据类型同样是流<字符串>。但是newStream只包含以“ca”开头的元素，因此由“cat”和“catch”这两个元素组成。

如果上面的代码片段中的 Lambda 表达式让你感到困惑，请记住，我们始终可以用具名函数重写 Lambda 表达式。现在就来重写上面的 Lambda 表达式，单词 -> word.startsWith("ca")。

第一步是确定这个 Lambda 表达式属于哪个函数式接口的实例。为此，我们查阅 filter()过滤操作 方法在 流 接口中的文档。文档中给出的 filter()过滤操作 方法签名如下：

流<T> filter(Predicate<? super T> predicate)

在我们更加熟悉流和 Lambda 表达式之前，最好先忽略类型通配符（“?”）。因此，我们可以把参数看作具有 Predicate<T> 数据类型。我们再次查阅 Java 官方文档，这次查找的是 Predicate<T>。正如预期的那样，它是一个函数式接口，这意味着它恰好有一个抽象方法。该方法的签名是 booleantest(T t)。请注意，我们的 filter()过滤操作 方法应用于一个字符串流，流<字符串>。因此，在这个示例中，类型参数 T 的值为 字符串。因此，要重写 Lambda 表达式 单词 -> word.startsWith("ca")，我们需要实现接口Predicate<String>。具体来说，我们需要实现方法 test(String t)，使其参数名为 单词，返回值为 word.startsWith("ca")。综合以上内容，我们得到以下代码。

```
class StartsWithCA implements Predicate<字符串> { public boolean test(String word) { return word.startsWith("ca"); } } StartsWithCA startsWithCA = new StartsWithCA(); 流<字符串> stream = Stream.of("bat", "cat", "bird", "mad", "catch", "ditch"); 流<字符串> newstream = stream.filter(startsWithCA);
```

获得新的过滤后的流 newstream后，我们可以对其执行进一步的计算。例如，我们可以计算元素数量：

```
流<字符串> newstream = stream.filter(word -> word.startsWith("ca")); long numStartWithCA = newstream.count(); // numStartWithCA = 2
```

不过，通常会将其写得更加简洁。我们不必为中间操作创建的每个流都定义一个新的局部变量，而是可以立即通过方法调用应用下一个操作：

```
long numStartWithCA = 流 .filter(word -> word.startsWith("ca")) .
count(); // numStartWithCA = 2
```

这与前面的代码片段等价，但熟悉这种语法后，它更加简洁易读。计算中的每个操作都写成以“.”开头的方法调用，并单独占一行，以提高可读性。

> 示例题 9。
>
> 在前一道示例题的解答基础上，编写一段代码，使用 Stream API 将 27 到 159（含）之间的<左右>整数相加。

3.2.4 foreach()

该 `foreach()` 操作会对输入流中的每个元素执行一个操作。它是终端操作，只应在计算结束时用于生成输出。切勿使用 `foreach()` 操作在中间操作中处理元素。就我们的用途而言，`foreach()` 的唯一用途是使用 `System.out.print()`和 `System.out.println()` 等方法输出流中的元素，如下例所示。

```
流<字符串> 流 = Stream.of("apple", "banana", "bagel"); stream.forEach(word ->
System.out.println(word));
```

此代码片段会生成以下输出

```
苹果 香
蕉 贝果
```

我们已经讨论过，如何始终将 Lambda 表达式重写为函数式接口的一个具名实例。还有另一种方式，可以重写某些简单的 Lambda 表达式。如果 Lambda 表达式除了调用一个已有名称的方法之外不执行任何操作，就可以使用 Java 的*方法引用*运算符“`::`”直接引用该方法。例如，代码片段`标准输出::println`等价于 Lambda 表达式`x -> 标准输出.println(x)`。方法引用常与 `foreach()`操作结合使用，如以下示例所示（该示例等价于前一个示例）。

```
流<字符串> stream5 = Stream.of("apple", "banana", "bagel");
stream5.forEach(System.out::println);
```

> 示例题 10。
>
> 编写一段使用 Stream API 将 27 到 159（含两端）的整数逐行打印出来的代码片段。

3.2.5 map()

`map()`操作用于对输入流中的每个元素应用一个方法。在数学表示法中，将函数f 映射到序列x_1、 x_2、x_3、 x_4.....的结果是$f(x_1)$， $f(x_2)$， $f(x_3)$， $f(x_4)$， .....。下面是一个使用 Java 流的示例。

```
流<字符串> 流 = Stream.of("apple", "banana", "bagel"); 流
    .map(单词 -> word.toUpperCase() + "***") .forEach(标准输
出::print);
```

输出为

```
APPLE***BANANA***BAGEL***
```

> 示例题 11。
>
> 假设文件 `info.txt`的每一行都存储一个非空字符串。编写一段代码，使用 Stream API 将文件中每一行的第一个字符分别打印在单独的行上。

3.2.6 mapToInt()

`mapToInt()`操作与 `map()`完全相同，只不过映射函数必须返回一个 `int`，因此生成的流是一个 `IntStream`。例如，我们可以找到 流<字符串> 的每个元素中字母“e”的位置，如下所示。

```
Stream<字符串> 流 = Stream.of("apple", "banana", "bagel"); 流
    .mapToInt(word -> word.indexOf('e')) .forEach(标准输出::
println);
```

输出为

```
4
-1
3
```

> 示例题 12。
>
> 编写一段代码，使用 Stream API 将文件`info.txt`中每一行的长度分别打印在单独的一行上。

3.2.7 reduce()

The `reduce()` 操作是一种终端操作，用于将流中的所有元素合并为单个输出。如果将此操作称为“combine”，或许会更容易理解。不过，“reduce”这个术语也很恰当，因为我们正在将整个流“归约”为单个输出。

The `reduce()` 操作比其他操作稍微复杂一些，因此我们通过一个示例来介绍它。暂时先不使用流，回到普通数组中的数据处理。假设我们有一个字符串数组，希望从每个字符串中取出第一个字符，并将这些字符合并成一个字符串。例如，数组 `{"apple", "banana", "bagel"}` 将生成输出“`abb`”。以下 Java 代码可以实现这一点。

```
字符串[] 数组 = { "apple", "banana", "bagel" }; 字符串 initialValue = ""; 字符串 resultSoFar = initialValue; 对于 (字符串 newElement: 数组) { resultSoFar = resultSoFar + newElement.charAt(0); }
```

这个算法的思路很明显。我们遍历数组，并将所有结果累积到变量 `resultSoFar`中。每次处理一个新元素时，我们都会将它与现有结果合并（在这个例子中，是添加 `newElement`的第一个字符），从而更新`resultSoFar`的值。我们还为结果指定了一个初始值，为了清晰起见，将其单独存储在变量 `initialValue`中。算法终止时，最终结果可以在累加器变量 `resultSoFar`中找到。

我们可以重构上面的代码片段，将累积结果的过程提取到单独的方法 `accumulate()`中，如下所示。

```
public static 字符串 accumulate(字符串 resultSoFar, 字符串 newElement) { return resultSoFar + newElement.charAt(0); } public static void main(String[] args) throws IOException { 字符串[] 数组2 = { "apple", "banana", "bagel" }; 字符串 initialValue = ""; 字符串 resultSoFar = initialValue; 对于 (字符串 newElement : array2) { resultSoFar = accumulate(resultSoFar, newElement); } 标准输出.println(resultSoFar); }
```

现在，我们可以使用上面示例中的术语来描述 `reduce()` 流操作。该操作有两个参数，分别对应于 `initialValue` 变量和 `accumulate()` 方法。因此，`reduce(initialValue, accumulate)` 流操作会使用给定的初始值进行初始化，然后依次对流中的每个元素应用 accumulate 方法。accumulate 方法通常由 Lambda 表达式指定。

回到我们这个具体示例，下面的代码展示了如何生成一个由流中每个元素的首字符组成的字符串。

```
流<字符串> 流 = Stream.of("apple", "banana", "bagel");       字符串 firstLetters
                    =         流.reduce( "", // 第一个参数是初始值，一个空字符串

(resultSoFar, newElement) -> resultSoFar + newElement.charAt(0)       // 第二个参数
（上文所示）是“累积”函数， // 以包含两个参数的 Lambda 表达式形式编写        );
```

请注意，表示累加器函数的 Lambda 表达式有两个参数，就像前面示例中的 `accumulate()` 方法一样：

```
(resultSoFar, newElement) -> resultSoFar + newElement.charAt(0)
```

请注意，我们学习的是 两个参数形式的 `reduce()` 操作。它还有单参数形式和三参数形式，但这里不作介绍。

示例题 13。

假设文件 `numbers.txt` 的每一行都存储一个以十进制记数法书写的数字。请编写一段代码，使用 Stream API 和 `java.math.BigDecimal` 类计算文件中所有数字的总和。该类可以确保处理小数时不会损失精度。（例如，进行金融计算时，采用这种方法就很重要。）

3.3 并行流

Java Stream API 的一项优势是，几乎无需费力就能将计算并行化。每个流都有一个内部设置，用于决定它以顺序方式还是并行方式运行。如果将流设为并行运行，Java 流库就会尝试把流拆分成多个块，并将它们交给各个线程同时处理。某些计算能从这种并行性中获益良多，因此可以更快完成。事实上，如果有 N 个 CPU 核心可用，理论上计算速度可以提升 N 倍（如果考虑超线程，甚至还能更快）。另一方面，有些计算无法从并行化中获益。由于设置并行流会产生额外开销，这类计算以并行模式运行时，实际上可能比顺序版本更慢。

对任何流应用 `parallel()` 操作，即可将其转换为并行流。应用 `sequential()` 操作，则可将流转换为顺序模式。以下代码示例展示了当数组 `values[]` 非常大时，利用并行性可以显著提升性能。我们假设存在一个布尔方法 `isPrime()`，它在参数为素数时返回 `true`。

```
int[] 值 = ...
素数个数 = IntStream.of(values)
                        .parallel()
                        .filter(n -> isPrime(n))
                        .count();
```

若要以顺序方式执行相同的计算，只需将“`parallel()`”替换为“`sequential()`”。程序 `ParallelPrimesDemo.java`（见下方代码清单）通过实测比较了这两种方法。

示例问题 14.

在自己的计算机上运行 `ParallelPrimesDemo.java`（见下方代码清单），并确定并行执行计算时的平均加速比。将其与计算机上可用的 CPU 核心数和/或超线程数相比如何？

3.4 程序清单：ParallelPrimesDemo.java

```
导入 java.util.Random;
导入 java.util.stream.IntStream;

/**
 * 这个类演示了使用并行流而非顺序流所能获得的加速效果
 * 通过使用并行流，而不是顺序流。
*/
公共类 ParallelPrimesDemo {

        // 一种刻意低效的判断整数是否为
        // n>=2 是素数的方法。我们只是想执行一项计算密集型的
        // 任务，以展示并行性的优势。
        private static boolean isPrime(int n) {
                for (int i = 2; i < n; i++) {
                        if (n % i == 0) {
                                返回 false;
                        }
                }
                返回 true;
        }

        public static void main(String[] args) {
                // 步骤 1. 创建一个包含大量随机整数的数组。我们确保
                // 每个整数都大于或等于 2，这样才有意义去判断
                // 该整数是否为素数。
                Random 随机数生成器 = new Random();
                最终 int 数值数量 = 2000000;
                最终 int 最大值 = 10000;
                int[] 数值 = new int[数值数量];
                for (int i = 0; i < 数值.length; i++) {
                        数值[i] = 2 + 随机数生成器.nextInt(最大值 - 2);
                }
```

```
                // 我们将剩余步骤重复执行几次，以便检查
                // 计时结果是否可靠。
                int 重复次数 = 5;
                for (int 重复次数 = 0; 重复次数 < 重复次数; 重复次数++) {

                        // 步骤 2。使用顺序流确定其中有多少个
                        // 随机整数是素数，并记录耗时。
                        long 素数数量 es1， 素数数量 es2;
                        long 开始时间，结束时间;
                        开始时间 = System.nanoTime();
                        素数数量1 = IntStream.of(values).sequential()
                                        .filter(n -> isPrime(n)).count();
                        结束时间 = System.nanoTime();
                        long 顺序执行时长 = (结束时间 - 开始时间)
                           / 1000000; // 毫秒

                        // 第 3 步。与上一步相同，但使用并行执行的
                        // 流。
                        开始时间 = System.nanoTime();
                        numPrimes2 = IntStream.of(values).并行执行()
                                        .filter(n -> isPrime(n)).count();
                        endTime = System.nanoTime();
                        长并行执行时长 = (结束时间 - 开始时间)
                           / 1000000; // 毫秒

                        // 结果最好相同!
                        断言 numPrimes1 == numPrimes2;

                        // 第 4 步：打印计时结果。
                        double 加速比 = (double) 顺序持续时间
                                        / 并行执行持续时间;
                        标准输出.format(
                        "顺序 %d毫秒，并行执行 %d毫秒，加速比 %2.2f\n",
                                  顺序执行时长、并行执行时长、加速比）；
                }
        }
}
```

4 关于函数式编程的总结

本章旨在介绍函数式编程中的一些理念。我们重点讨论了将函数用作一等对象，也就是说，函数可以赋值给局部变量、作为参数传递，以及由其他函数返回。然而，这只是函数式编程的一个方面。编写纯函数式程序时，另一个关键概念是不可变性。这指的是对象一经创建，其状态便永远不会改变。有些 Java 对象具有这一特性。例如，Java字符串对象和流对象本质上都是不可变的。但在纯函数式程序中，一切都是不可变的。因此，函数没有副作用。这与 Java 形成鲜明对比，在 Java 中，函数（即方法）经常会改变对象的状态。例如，`setColor(Color c)`这类修改器方法会改变`java.awt.Graphics`类中调用该方法的`Graphics`对象的状态。其副作用是，后续调用绘图

诸如 `drawRect()`之类的方法会使用不同的颜色进行绘制。在纯函数式编程中，不存在副作用。这可以让正确编写和调试代码变得更加容易，尤其是对于多线程代码而言。总而言之，在本章中，我们只学习了函数式编程的一个重要方面：Lambda 表达式。要充分理解函数式范式，建议学习一门功能完备的函数式编程语言，例如 Haskell、Lisp、Scheme 或 F#。

5 个示例题的解答

示例题 1。

```
applyToMinusNine(add5) → -4
applyToMinusNine(multBy3IfPositive) → 0
```

示例问题 2。

```
applyToMinusNine(add5) → -4
applyToMinusNine(multBy3IfPositive) → 0
```

示例问题 3。

```
applyToSeven(lambda potato: potato%4 + potato*potato) → 52
applyToMinusNine(lambda oak: math.factorial(oak+12) * oak) → -54
```

示例问题 4。

```
val5 = 56
val6 = -90
```

示例题 5。

```
(u,v,w)->Math.sqrt(u*u + v*v + w*w)
```

示例问题 6。

有很多种方法可以实现这一点。以下是一些示例：

(i) 流< Double> doubleObjects = Stream.of(23.4, 69.7, -25.88, 31.3363) ;

（ii）`DoubleStream doublePrimitives = DoubleStream.of(23.4, 69.7, -25.88,31.3363);`

（iii）以下任一种方法都适用于此处：IntStream 范围1 = IntStream.range(100, 201); IntStream 范围2 = `IntStream.rangeClosed(100, 200);`

示例题 7。

流<字符串> gatsby = Files.lines(路径.get("GreatGatsby.txt")); long numLines = gatsby.count();

示例问题8。

int sum27to159 = IntStream.闭区间范围(27, 159).sum()求和操作;

示例题 9。

int sumOdd27to159 = IntStream.闭区间范围(27, 159) .filter(x -> x % 2 == 1) .sum()求和操作;

示例题 10。

IntStream.闭区间范围(27, 159).forEach(标准输出::println);

示例题 11。

流<字符串> info = Files.lines(Paths.get("info.txt")); info.map(s-> s.substring(0, 1)).forEach(标准输出::println);

示例题 12。

流<字符串> info = Files.lines(Paths.get("info.txt")); info.mapToInt(s-> s.length()).forEach(标准输出::println);

示例题 13。

流<字符串> numbers = Files.lines(Paths.get("numbers.txt")); BigDecimal sum = numbers .map(s -> new BigDecimal(s)) .reduce(new BigDecimal(0), (tot, val) -> tot.add(val)); System.out.println(sum);

示例问题 14。

以下是在英特尔酷睿 i5-8350U 处理器上获得的结果：

> 顺序 2907ms，并行执行 634ms，加速比 4.59 顺序 2790ms，并行执行 639ms，加速比 4.37 顺序 2714ms，并行执行 651ms，加速比 4.17 顺序 2729ms，并行执行 651ms，加速比 4.19 顺序 2731ms，并行执行 646ms，加速比 4.23

英特尔称，这款处理器拥有 4 个核心和 8 个线程。对上述加速比的一种合理解释是，四个核心中的每一个都能够同时完成部分处理，而这款处理器的超线程技术则带来了一些额外的性能提升。