第3章

流

Java 8 中新增的特性旨在帮助程序员写出更好的代码，其中对核心类库的改进是很关键的一部分，也是本章的主要内容。对核心类库的改进主要包括集合类的 API 和新引入的流（Stream）。流使程序员得以站在更高的抽象层次上对集合进行操作。

本章会介绍 `Stream` 类中的一组方法，每个方法都对应集合上的一种操作。

3.1 从外部迭代到内部迭代

本章及本书其余部分的例子大多围绕 1.3 节介绍的案例展开。

Java 程序员在使用集合类时，一个通用的模式是在集合上进行迭代，然后处理返回的每一个元素。比如要计算从伦敦来的艺术家的人数，通常代码会写成例 3-1 这样。

例 3-1　使用 `for` 循环计算来自伦敦的艺术家人数

```
int count = 0;
for (Artist artist : allArtists) {
    if (artist.isFrom("London")) {
        count++;
    }
}
```

尽管这样的操作可行，但存在几个问题。每次迭代集合类时，都需要写很多样板代码。将

for 循环改造成并行方式运行也很麻烦，需要修改每个 for 循环才能实现。

此外，上述代码无法流畅传达程序员的意图。for 循环的样板代码模糊了代码的本意，程序员必须阅读整个循环体才能理解。若是单一的 for 循环，倒也问题不大，但面对一个满是循环（尤其是嵌套循环）的庞大代码库时，负担就重了。

就其背后的原理来看，for 循环其实是一个封装了迭代的语法糖，我们在这里多花点时间，看看它的工作原理。首先调用 iterator 方法，产生一个新的 Iterator 对象，进而控制整个迭代过程，这就是*外部迭代*。迭代过程通过显式调用 Iterator 对象的 hasNext 和 next 方法完成迭代。展开后的代码如例 3-2 所示，图 3-1 展示了迭代过程中的方法调用。

例 3-2　使用迭代器计算来自伦敦的艺术家人数

```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

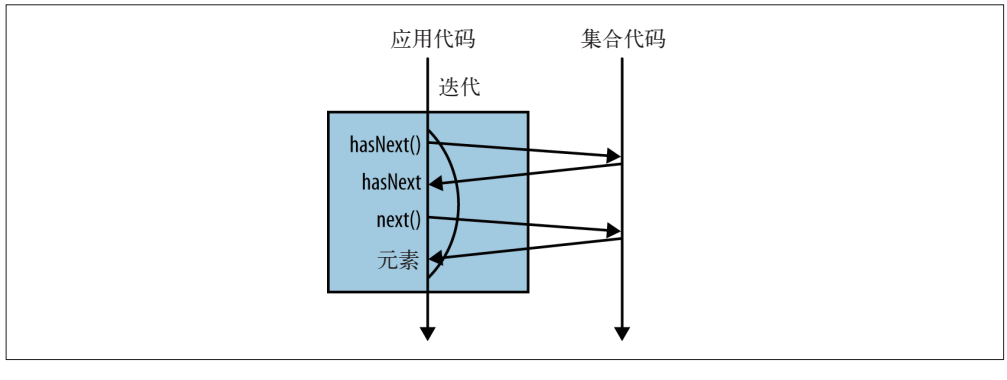

图 3-1：外部迭代

然而，外部迭代也有问题。首先，它很难抽象出本章稍后提及的不同操作；此外，它从本质上来讲是一种串行化操作。总体来看，使用 for 循环会将行为和方法混为一谈。

另一种方法就是*内部迭代*，如例 3-3 所示。首先要注意 stream() 方法的调用，它和例 3-2 中调用 iterator() 的作用一样。该方法不是返回一个控制迭代的 Iterator 对象，而是返回内部迭代中的相应接口：Stream。

例 3-3　使用内部迭代计算来自伦敦的艺术家人数

```
long count = allArtists.stream()
                       .filter(artist -> artist.isFrom("London"))
                       .count();
```

图 3-2 展示了使用类库后的方法调用流程，与图 3-1 形成对比。

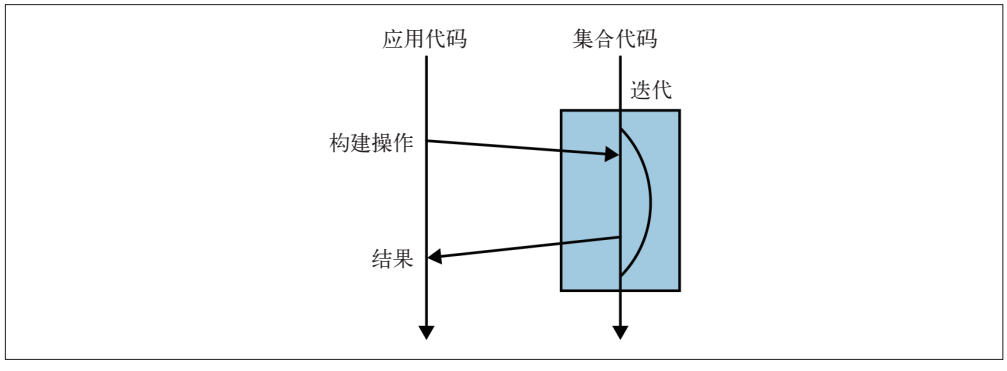

图 3-2：内部迭代

Stream 是用函数式编程方式在集合类上进行复杂操作的工具。

例 3-3 可被分解为两步更简单的操作：

- 找出所有来自伦敦的艺术家；
- 计算他们的人数。

每种操作都对应 Stream 接口的一个方法。为了找出来自伦敦的艺术家，需要对 Stream 对象进行过滤：filter。过滤在这里是指“只保留通过某项测试的对象”。测试由一个函数完成，根据艺术家是否来自伦敦，该函数返回 true 或者 false。由于 Stream API 的函数式编程风格，我们并没有改变集合的内容，而是描述出 Stream 里的内容。count() 方法计算给定 Stream 里包含多少个对象。

3.2 实现机制

例 3-3 中，整个过程被分解为两种更简单的操作：过滤和计数，看似有化简为繁之嫌——例 3-1 中只含一个 for 循环，两种操作是否意味着需要两次循环？事实上，类库设计精妙，只需对艺术家列表迭代一次。

通常，在 Java 中调用一个方法，计算机会随即执行操作：比如，System.out.println("Hello World"); 会在终端上输出一条信息。Stream 里的一些方法却略有不同，它们虽是普通的 Java 方法，但返回的 Stream 对象却不是一个新集合，而是创建新集合的配方。现在，尝试思考一下例 3-4 中代码的作用，一时毫无头绪也没关系，稍后会详细解释。

例 3-4　只过滤，不计数

```
allArtists.stream()
          .filter(artist -> artist.isFrom("London"));
```

这行代码并未做什么实际性的工作，filter 只刻画出了 Stream，但没有产生新的集合。像 filter 这样只描述 Stream，最终不产生新集合的方法叫作*惰性求值方法*；而像 count 这样最终会从 Stream 产生值的方法叫作*及早求值方法*。

如果在过滤器中加入一条 println 语句，来输出艺术家的名字，就能轻而易举地看出其中的不同。例 3-5 对例 3-4 作了一些修改，加入了输出语句。运行这段代码，程序不会输出任何信息！

例 3-5　由于使用了惰性求值，没有输出艺术家的名字

```
allArtists.stream()
          .filter(artist -> {
              System.out.println(artist.getName());
              return artist.isFrom("London");
           });
```

如果将同样的输出语句加入一个拥有终止操作的流，如例 3-3 中的计数操作，艺术家的名字就会被输出（见例 3-6）。

例 3-6　输出艺术家的名字

```
long count = allArtists.stream()
                       .filter(artist -> {
                           System.out.println(artist.getName());
                           return artist.isFrom("London");
                       })
                       .count();
```

以披头士乐队的成员作为艺术家列表，运行上述程序，命令行里输出的内容如例 3-7 所示。

例 3-7　显示披头士乐队成员名单的示例输出

```
John Lennon
Paul McCartney
George Harrison
Ringo Starr
```

判断一个操作是惰性求值还是及早求值很简单：只需看它的返回值。如果返回值是 Stream，那么是惰性求值；如果返回值是另一个值或为空，那么就是及早求值。使用这些操作的理想方式就是形成一个惰性求值的链，最后用一个及早求值的操作返回想要的结果，这正是它的合理之处。计数的示例也是这样运行的，但这只是最简单的情况：只含两步操作。

整个过程和*建造者*模式有共通之处。建造者模式使用一系列操作设置属性和配置，最后调用一个 build 方法，这时，对象才被真正创建。

读者一定会问：“为什么要区分惰性求值和及早求值？”只有在对需要什么样的结果和操

作有了更多了解之后，才能更有效率地进行计算。例如，如果要找出大于 10 的第一个数字，那么并不需要和所有元素去做比较，只要找出第一个匹配的元素就够了。这也意味着可以在集合类上级联多种操作，但迭代只需一次。

3.3 常用的流操作

为了更好地理解 Stream API，掌握一些常用的 Stream 操作十分必要。除此处讲述的几种重要操作之外，该 API 的 Javadoc 中还有更多信息。

3.3.1 collect(toList())

collect(toList()) 方法由 Stream 里的值生成一个列表，是一个及早求值操作。

Stream 的 of 方法使用一组初始值生成新的 Stream。事实上，collect 的用法不仅限于此，它是一个非常通用的强大结构，第 5 章将详细介绍它的其他用途。下面是使用 collect 方法的一个例子：

```
List<String> collected = Stream.of("a", "b", "c")  ❶
                               .collect(Collectors.toList());  ❷

assertEquals(Arrays.asList("a", "b", "c"), collected);  ❸
```

这段程序展示了如何使用 collect(toList()) 方法从 Stream 中生成一个列表。如上文所述，由于很多 Stream 操作都是惰性求值，因此调用 Stream 上一系列方法之后，还需要最后再调用一个类似 collect 的及早求值方法。

这个例子也展示了本节中所有示例代码的通用格式。首先由列表生成一个 Stream ❶，然后进行一些 Stream 上的操作，继而是 collect 操作，由 Stream 生成列表❷，最后使用断言判断结果是否和预期一致❸。

形象一点儿的话，可以将 Stream 想象成汉堡，将最前和最后对 Stream 操作的方法想象成两片面包，这两片面包帮助我们认清操作的起点和终点。

3.3.2 map

如果有一个函数可以将一种类型的值转换成另外一种类型，map 操作就可以使用该函数，将一个流中的值转换成一个新的流。