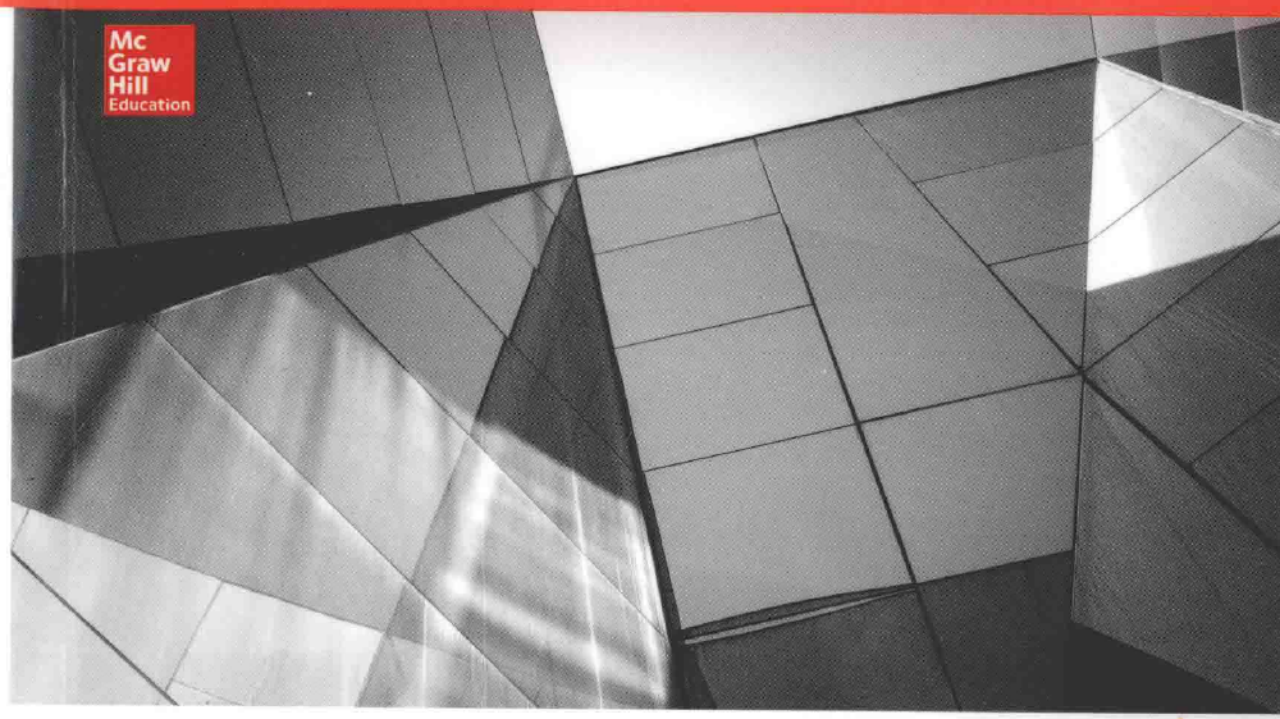
ORACLE

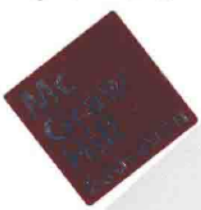


Mastering Lambdas: Java Programming in a Multicore World

Java

精通lambda表达式：Java多核编程

使用lambda表达式和流的最佳实践



[美] Maurice Naftalin 著
张 龙 译

清华大学出版社

精通 lambda 表达式：Java 多核编程

[美] Maurice Naftalin 著

张 龙 译

清华大学出版社

北 京

Maurice Naftalin
Mastering Lambdas: Java Programming in a Multicore World
EISBN：978-0-07-182962-5

图书在版编目(CIP)数据

精通 lambda 表达式：Java 多核编程 /(美) 那夫特林(Naftalin,M.) 著；张龙 译. —北京：清华大学出版社，2015
书名原文：Mastering Lambdas: Java Programming in a Multicore World
ISBN 978-7-302-40553-5

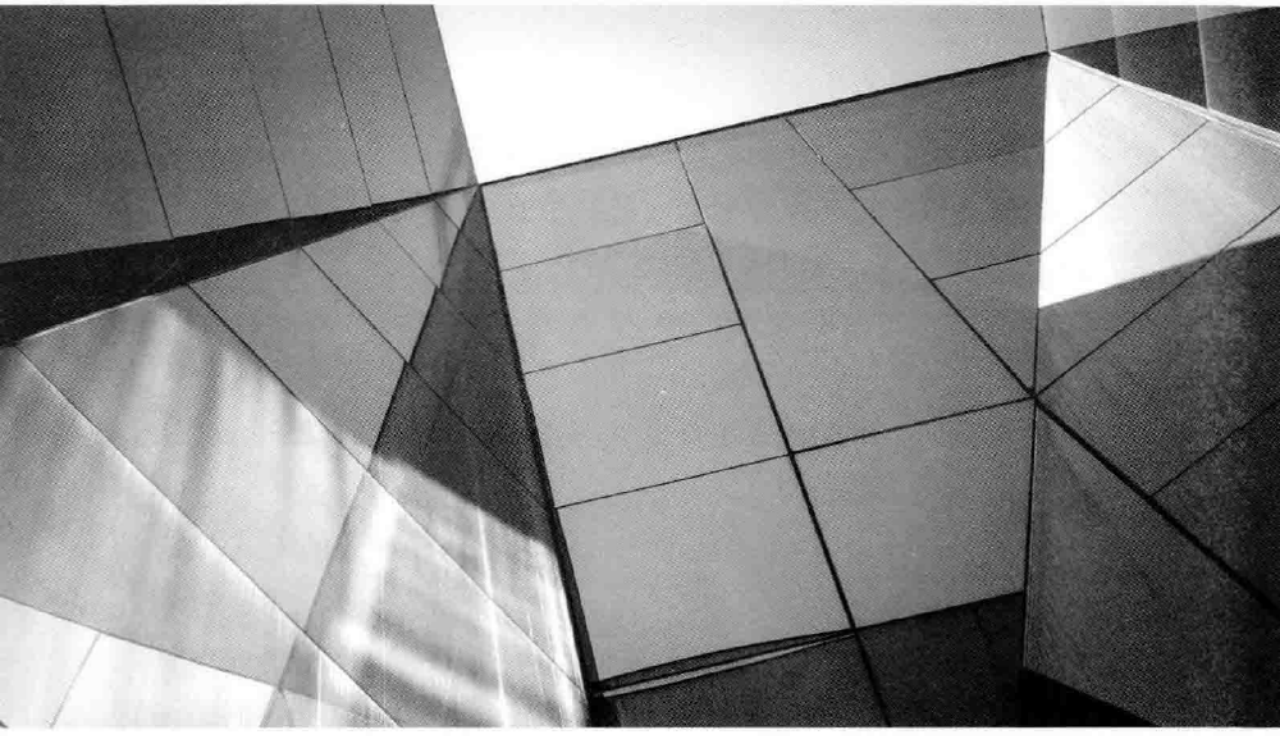
Ⅰ. ①精… Ⅱ. ①那… ②张… Ⅲ. JAVA 语言—程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2015)第 137442 号

责任编辑：王　军　于　平
装帧设计：孔祥峰
责任校对：曹　阳
责任印制：沈　露

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京密云胶印厂
经　　销：全国新华书店
开　　本：148mm×210mm　　印　　张：7.5　　字　　数：181 千字
版　　次：2015 年 8 月第 1 版　　印　　次：2015 年 8 月第 1 次印刷
印　　数：1～2000
定　　价：39.00 元

产品编号：062758-01



译 者 序

2014 年 3 月，万众期待的 Java 8 终于发布了。Java 8 可谓 Java 语言历史上变化最大的一个版本，这体现在语言、库与虚拟机的增强上。Java 语言现在已经渗透到软件行业的各个领域，从大数据领域的 Hadoop 到企业级开发的 Spring 框架，再到移动领域的 Android，无处不显现 Java 的身影。作为一门有着近 20 年历史的语言，Java 影响了整整一代程序员。随着时代的变迁，动态语言、函数式编程越来越受到广大开发人员的关注，从这个角度来看，Java 似乎有些臃肿和笨重。但强大的 JVM 赋予了 Java 新的生命力，有越来越多的语言能够运行在 Java 虚拟机上并与 Java 交互，如 Groovy、Scala、Clojure 等，这些语言反过来又促进了 Java 的

持续发展。

Java 8 引入了诸多令人心潮澎湃的新特性，如 lambda 表达式、Stream API、并发库的增强、默认方法、函数式接口、方法与构造方法引用、新的日期与时间 API、注解增强等，这些新特性促进了 Java 的持续发展。在 Java 8 的诸多特性当中，lambda 表达式是最为人所关注的。借助于 lambda 表达式，可以编写出性能更好、可读性更强、可维护性更好的代码，本书就是一本专门介绍 Java lambda 表达式的著作。

本书共分为 7 章，按照由浅入深、循序渐进的方式对 Java lambda 表达式进行了深入的剖析与介绍。首先从 Java 8 新特性开始，对 lambda 表达式作一简要说明；接下来介绍 lambda 表达式的基础，让读者对 lambda 表达式有个感性的认识；然后对流与管道进行深入的讲解，同时剖析起始流、终止流，并对流的性能做了详尽的阐述；最后，介绍默认方法的含义及其对于 API 演化的重要意义。本书不仅介绍 Java lambda 表达式的基础知识，还深入讲解了构成 lambda 表达式的重要组件，并对其适用场景进行了广泛的说明。

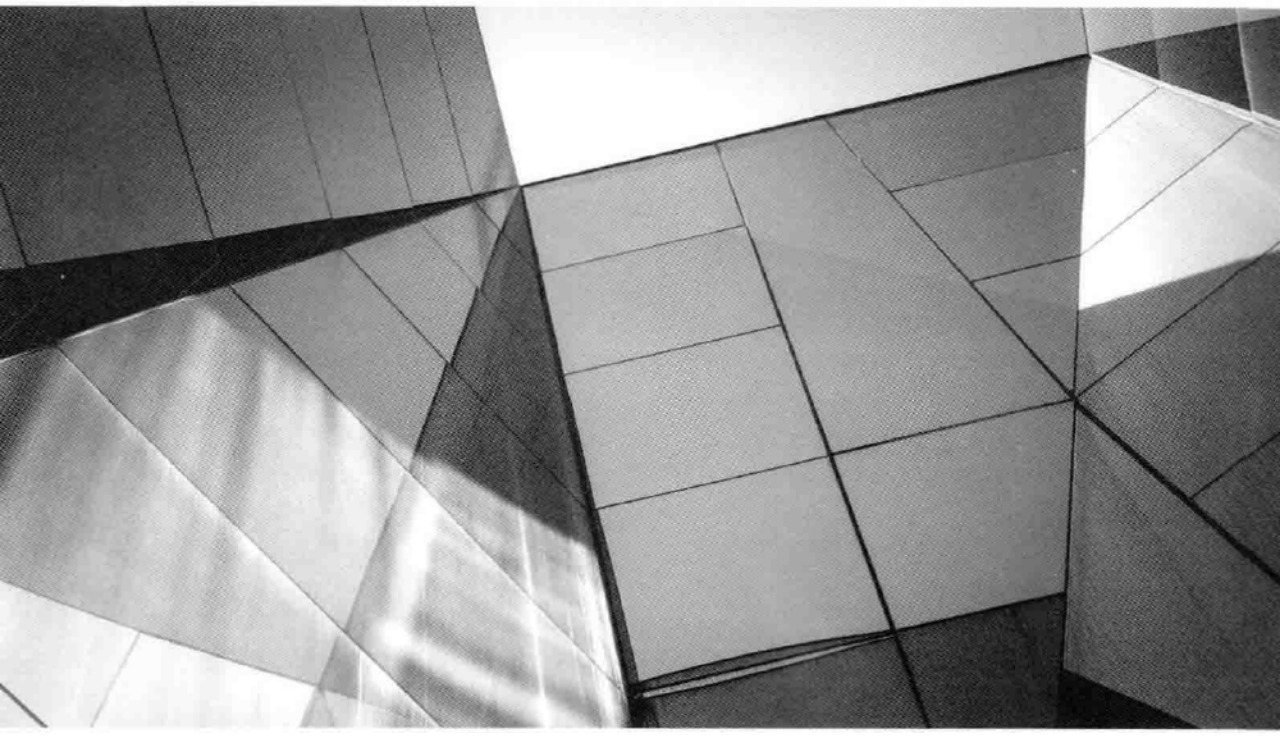
虽然现在很多公司开发所用的 JDK 尚未升级到最新的 JDK 8，但学习并实践 Java 8 的新特性是每一个有追求的 Java 开发者都应该做的事情，因为这是未来 Java 的发展方向。熟练掌握并灵活运用 lambda 表达式可以帮助你编写出更加优雅的代码，只需要寥寥数行代码即可解决之前十几行、甚至几十行代码才能完成的工作。但不得不提的是，要想熟练掌握 lambda 表达式并不是一件简单轻松的工作，因为其背后有很多重要理论需要学习者充分理解，而本书则可以帮助你更快、更好地掌握 lambda 表达式，因为书中所介绍的内容不仅会让你知其然，更会让你知其所以然。学习完本书后，我相信每一位读者都能够流畅地使用 lambda 表达式开始全

新 Java 代码的编写。

翻译技术图书是艰苦的劳动，不仅有体力的付出，更有大量脑力的付出。在此，请容许我感谢清华大学出版社的编辑，非常感谢他们在图书翻译过程中对我的包容与鼓励；其次，我要感谢我的父母，没有你们的精心抚育与教诲就不会有今天的我；再次，我将深深的感激之情送给我的妻子张明辉，谢谢你在图书翻译过程中给予我持续不断的支持与理解，本书的顺利出版有你一半的功劳；最后，将本书送给我亲爱的孩子张梓轩小朋友，愿你未来能够开开心心，茁壮成长。

本书的所有章节由张龙翻译，参加本次翻译工作的还有张彤辉、焦伟、张淑贤、李志芹、张兴国、马元贺、王辉、王凯、单会明、周锐、王冠、高鹏、张淑华，在此一并表示感谢。

尽管在翻译过程中译者已经尽了最大的努力以确保译文的准确和流畅，但囿于水平有限，读者在阅读过程中如果发现错误或不准确之处请及时与我联系，以便再版时修正。我的邮箱是 zhanglong217@163.com，博客是 http://blog.csdn.net/ricohzhanglong，新浪微博是@风中叶的思考，欢迎关注。最后，祝各位读者阅读愉快，早日掌握 lambda 表达式这一利器，编写出更加优雅的 Java 代码。



作 者 简 介

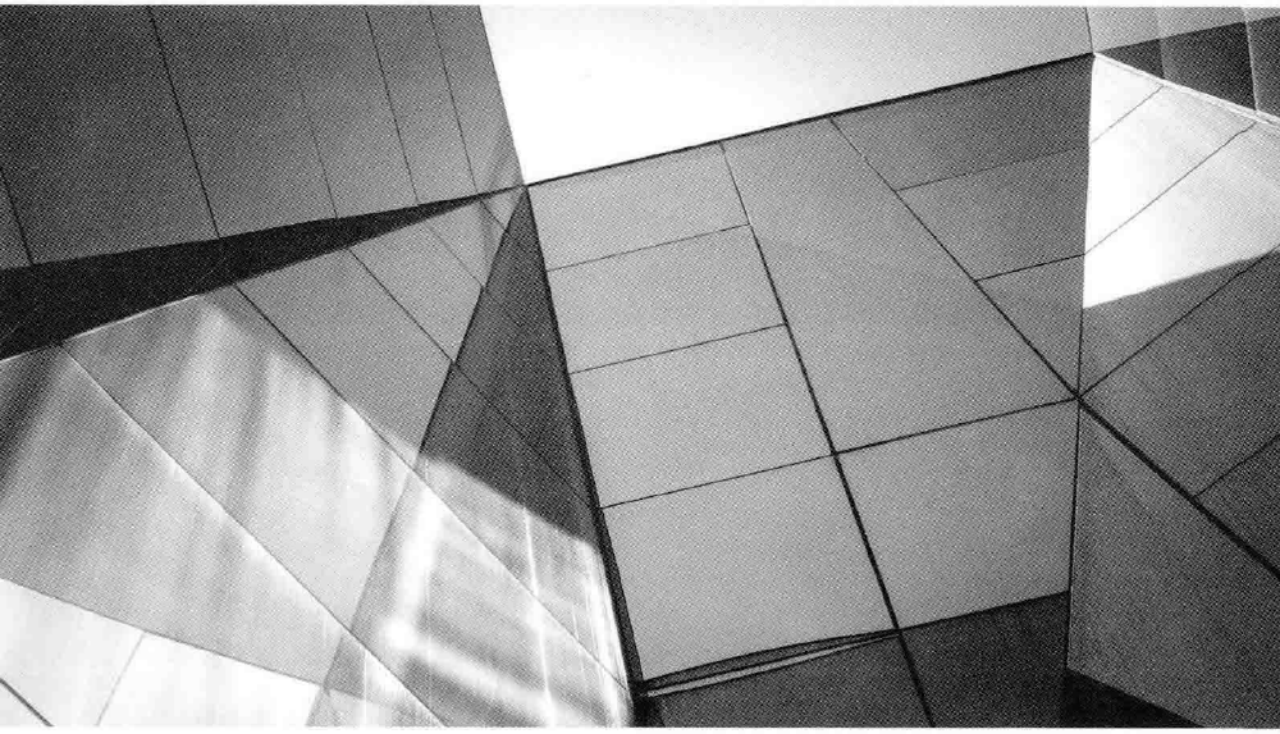
Maurice Naftalin 在 IT 领域拥有 30 多年的经验，担任过开发者、设计师、架构师、经理、教师以及作者等角色。Naftalin 是经过认证的 Java 程序员，使用过 Java 的各个发布版本。他在 Java 与业务上的经历让他对 Java SE 8 中引入 lambda 表达式所带来的根本性变化有着独到的见解。Naftalin 是各种大会上的常客，包括一年一度的 JavaOne。他与 Oracle 开发团队协作运营着一个颇受欢迎的网站——www.lambdafaq.org，该网站主要关注于 Java 8 中的新语言特性。

技术编辑简介

Stuart Marks 就职于 Oracle Java 平台组的 JDK 核心库团队。他目前从事的是 lambda、流与集合相关的工作，同时改进测试质量与性能。他之前在 Sun Microsystems 从事 JavaFX 与 Java ME 相关的工作。他在窗口系统、交互式图形以及移动和嵌入式系统领域有着 20 多年的软件平台产品开发经验。Stuart 拥有斯坦福大学计算机科学硕士学位以及电子工程学士学位。他目前与妻子和女儿居住在加利福尼亚。

Brian Goetz 是 Java 编程的权威之一。他是畅销书 *Java Concurrency in Practice* 的作者，此外还撰写了 75 篇之多的关于 Java 开发的文章。他是 JSR-335(Java 语言 lambda 表达式)规范的

制定领头人，并且在很多 JCP 专家组中担任职位。Brian 是 Oracle 的 Java 语言架构师。

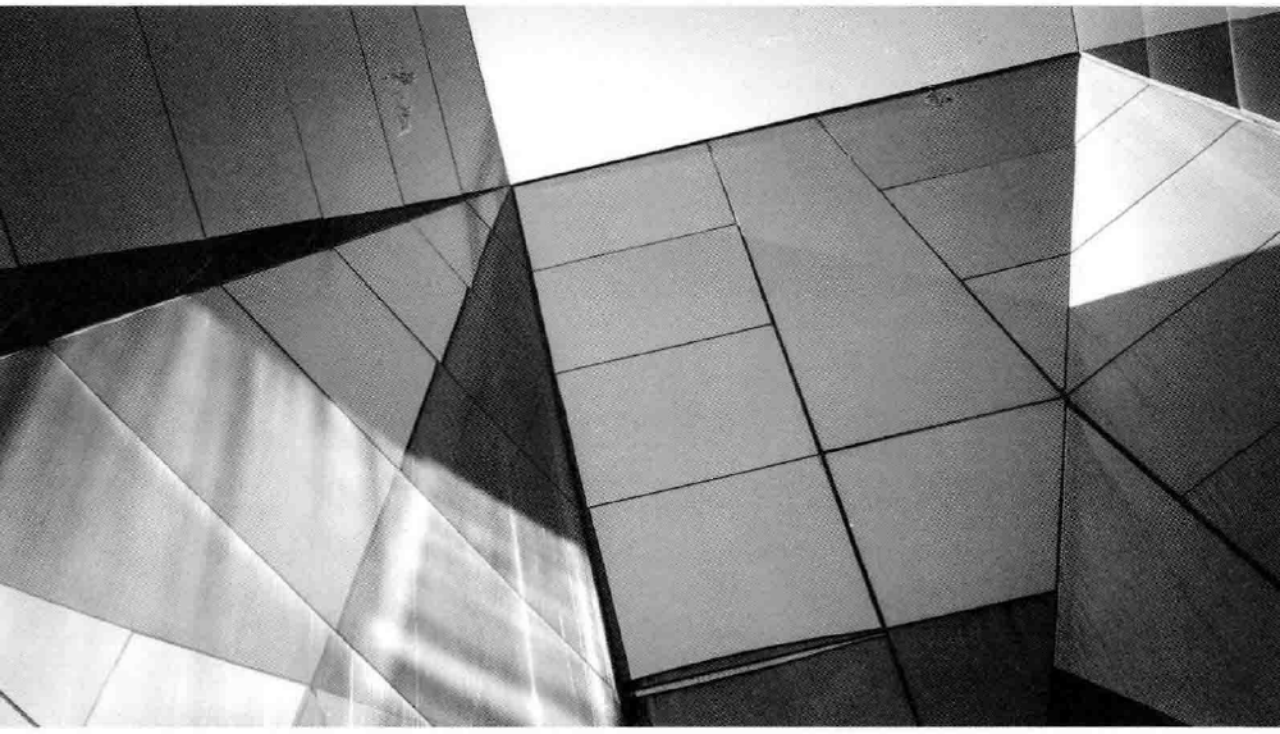


致　谢

如果没有来自于 Oracle 语言团队的伙伴们(Brian Goetz、Paul Sandoz、Aleksey Shipilev 与 Dan Smith)持续不断的帮助、鼓励与反馈，本书是不可能面世的。Stuart Marks 给出的建议非常有价值，对打磨本书起到了重要作用。

我要感谢 Graham Allan、Maurizio Cimadore、Chris Czarnecki、John Kostaras、Kirk Pepperdine、Jeremy Prime 与 Philip Wadler，他们的审校避免了很多错误的发生，同时经常给我指出新的方向。当然，任何遗留的错误都是我的责任。

非常感激本书的编辑 Brandi Shailer，感谢她在本书漫长的写作过程中给予我的耐心和乐观态度。



序　言

Java SE 8 是有史以来对 Java 语言和库改变最大的一次。既然你手头阅读的这本书的书名是《精通 lambda 表达式：Java 多核编程》，那么你可能已经知道了最大的新特性就是增加了 lambda 表达式。根据视角的不同，这种演变可能起始于 2009 年(那时启动了 Project lambda)，也可能起始于 2006 年(有几个提案希望 Java 增加闭包)，还有可能起始于 1997 年(引入了内部类)，甚至起始于 1941 年(Alonzo Church 发布了关于计算理论的基础研究工作成果，lambda 表达式这个名字就是从中得来的)。

无论多久，这都是关于时间的问题！虽然一开始 lambda 表达式似乎只是“另一个语言特性”而已，但实际上，它们会改变你

思考编程的方式，提供强大的新工具，将抽象应用到你每天都会面对的编程挑战中。当然，Java 已经提供了用于抽象的强大工具，例如继承与泛型等，但它们在很大程度上只是关于数据抽象的。lambda 表达式则提供了用于对行为进行抽象的更棒的工具来弥补这一点。

在拥抱 lambda 表达式的过程中，Java 针对函数式编程采取了一种温和的转变方式。虽然看起来面向对象编程与函数式编程之间是竞争关系，却为我们提供了互补的工具来管理程序的复杂性。随着硬件并行化持续升温，函数式编程的构建块(不变值与纯函数)甚至成为管理这种复杂性的更为有效的工具。

《精通 lambda 表达式：Java 多核编程》以分层、有序的方式详细介绍了 Java SE 8 中关于语言与库的新特性，包括 lambda 表达式、默认方法与 Streams 库等。更为重要的是，它将特性的细节与底层的设计决策联系起来，促使读者能够理解背后的动机与原则，从而最大限度地掌握这些新特性。与此同时，本书专注于真正的价值，这不是特性本身，而是特性所带来的好处：表述性更强、更为强大且不易出错的用户代码。让读者意识到这种好处是本书真正所关注的方面。

让本书指引你利用全新且改进的Java进行编程吧。一旦起步，我相信你就再也停不下来了！

——Brian Goetz

Oracle 公司 Java 语言架构师

前　言

Java 8 可谓 Java 语言历史上变化最大的一个版本，其承诺要调整 Java 编程向着函数式风格迈进，这有助于编写出更为简洁、表达力更强，并且在很多情况下能够利用并行硬件的代码。在本书中，你将会发现引入 lambda 表达式这一表面上看起来细小的变化将如何使这一切成为可能。你将学习到如何通过 lambda 表达式使用一行代码编写 Java 函数，如何通过这种功能使用新的 Stream API 进行编程，如何将冗长的集合处理代码压缩为简单且可读性更好的流程序。学习创建和消费流的机制，分析其性能，能够判断何时应该调用 API 的并行执行特性。

最后，为将新特性集成到现有的 Java 平台库中，需要对已有

的集合接口进行演化，而之前由于兼容性问题这一点是没法实现的。你将学习到如何通过默认方法来解决这些问题，如何在演化自己的 API 时使用它们。

第 1 章　走进新生代的 Java

本章为将 lambda 表达式与流引入到 Java 中做好了准备，其变化的动机是需要更好的编程模型以及让Java 开始为多核处理器提供支持。

第 2 章　Java lambda 表达式基础

本章介绍了 lambda 表达式的语法，如何使用它们，在何处使用及其与匿名内部类的区别，以及由方法和构造器引用所提供的便捷缩写。

第 3 章　流与管道介绍

本章介绍了流的生命周期以及流编程的基础知识，提供了通过流源以及中间和终止操作处理集合的示例。

第 4 章　终止流：集合与汇聚

本章详细介绍了终止操作，特别是如何通过可变的汇聚操作将流元素汇聚到集合中。本章通过收集器(可变汇聚的库实现)扩展了第 3 章的示例。我们将会看到何时应该超越库实现的限制，以及如何编写自己的收集器。

第 5 章　起始流：源与分割迭代器

本章介绍了起始流的各种方式，包括使用库类，以及在必要时编写自己的分割迭代器。本章还深入介绍了流编程中的异常处

理。通过流处理重新实现 grep 的各种选项来展现出该模型的灵活性。

第 6 章　流的性能

本章介绍了如何确定并行执行的流处理的相对性能，方式是将源、中间操作的负载以及终止操作的并发性分割开来进行度量。此外还引入了微基准测试度量流的性能，同时还通过这些方式对书中的其他程序进行了分析。

第 7 章　使用默认方法来演化 API

本章介绍了新引入的默认方法是如何解决Java编程中长久以来存在的问题的，特别是如何首次使得基于接口的 Java API 的演化成为可能。本章还介绍了静态接口方法的使用。

本书读者对象

本书面向那些使用过 Java 5 及之前任意版本，同时又听说过 Java 8 中激动人心的变化，并且想要学习它们的 Java 开发者。你无须了解其他语言中的 lambda 表达式与闭包，也无须拥有函数式编程经验(当然，如果知道会更好)。

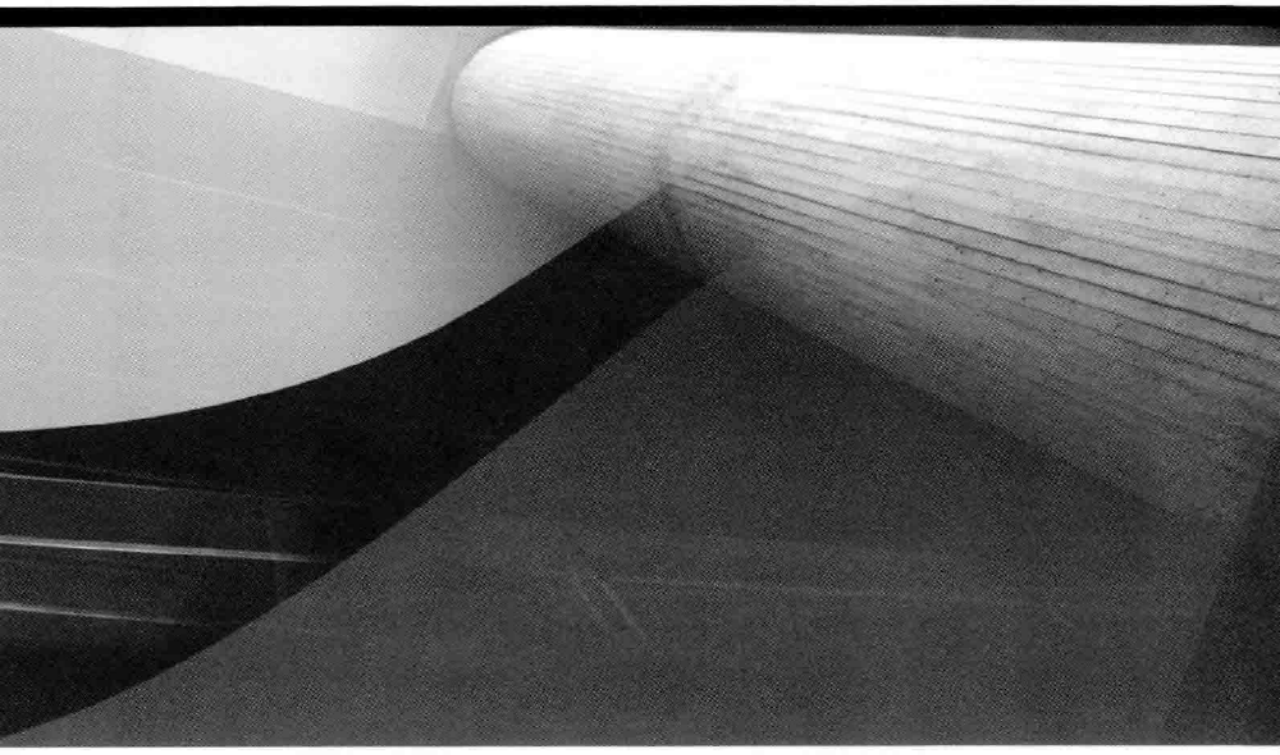
除了 Java 集合框架的标准集合外，本书不要求你熟悉其他的平台库，如果对标准集合不熟悉，请适时参阅 Javadoc 文档。

某些章节提供了一些高级主题：它们适合于延伸阅读。

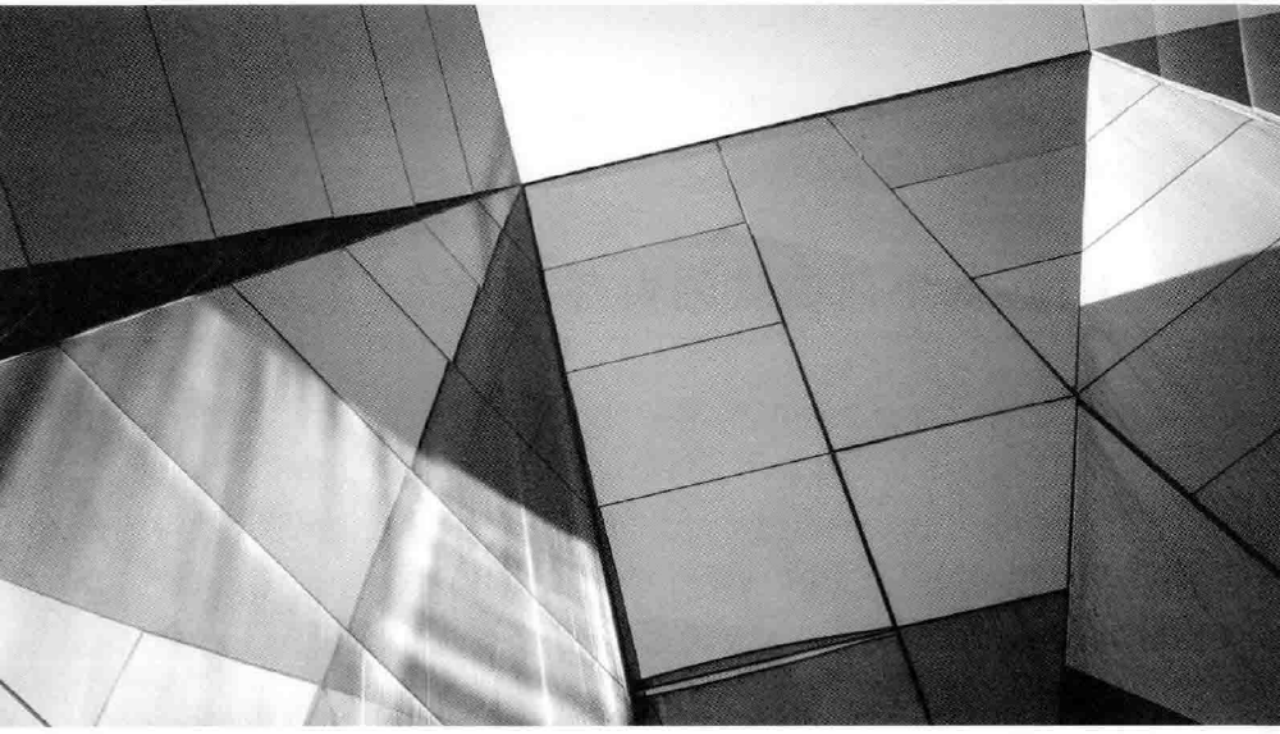
示例、反馈与进一步学习

书中的代码可以从Oracle出版社的网站下载，网址是www.OraclePressBooks.com。源代码与勘误也位于本书的产品页www.mhprofessional.com。只需要搜索ISBN并下载必要的文件即可。

读者可以访问本书的支持网站 www.masteringlambdas.org 进行讨论、寻找进一步学习的链接，还可以联系作者。



目　录



第1章

走进新生代的 Java

Java 8 可谓 Java 语言历史上变化最大的一个版本，这体现在语言、库与虚拟机协调一致的变化上。其承诺要改变我们思考 Java 程序执行的方式，并且让语言适合于不久之后将要到来的在大规模并行硬件上的使用。不过相对于如此重要的创新，语言的实际变化却显得有些微不足道。这些表面上的微小变化到底是如何产生这种巨大的差别的呢？我们为何要改变Java中已经使用了这么多年(实际时间比这还要长)的编程模型呢？本章将会谈及这种模

型的限制，同时还会介绍 Java 8 中 lambda 相关的特性是如何促使 Java 不断演进以满足新一代硬件架构的挑战的。

1.1 从外部迭代到内部迭代

首先来看一段简单的代码，这段代码会迭代一个包含着可变对象的集合，并对集合中的每个对象调用一个方法。如下代码片段会构造一个 java.awt.Point(Point 是个便捷、简单的库类，包含了一对(x,y)坐标)对象的集合。接下来，代码会对集合进行迭代，并对每个 Point 沿着 x 与 y 轴各平移 1 个单位的距离。

```
    List<Point> pointList = Arrays.asList(new Point(1, 2), new
Point(2, 3));
    for (Point p : pointList) {
       p.translate(1, 1);
    }
```

在 Java 5 引入 for-each 循环前，我们可以像下面这样编写循环：

```
    for (Iterator pointItr = pointList.iterator();
pointItr.hasNext(); ) {
        ((Point) pointItr.next()).translate(1, 1);
    }
```

下面这种写法会更清晰一些(不过并不推荐这种写法，因为它扩大了 pointItr 的作用域)：

```
    Iterator pointItr = pointList.iterator();
    while (pointItr.hasNext()) {
       ((Point) pointItr.next()).translate(1, 1);
    }
```

上述代码中，pointList 会创建一个 Iterator 对象，接下来我们通过该对象依次访问 pointList 中的元素。这个版本的代码还是非常有意义的，因为时至今日这正是 Java 编译器生成的用于实现 for-each 循环的代码。对于我们来说，其关键在于访问 pointList 中元素的顺序是由 Iterator 控制的——我们无法改变这一点。比如，针对 ArrayList 的 Iterator 会按照先后顺序返回列表中的元素。

这么做为什么会有问题呢？毕竟，Java Collections Framework 是在 1998 年设计的，看起来按照这种方式来指定列表元素的访问顺序是合情合理的。从那时起发生了什么变化呢？

部分原因在于硬件在不断演进。长久以来，工作站与服务器都配有多个处理器，不过在设计 Java Collections Framework 框架的 1998 年与个人电脑中首个双核处理器出现的 2005 年之间，芯片设计领域的革命爆发了。40 年来处理器速度以指数级别增长的趋势停止了，这是由于如下不可避免的事实造成的：信号泄漏、散热不充分，以及虽然到达了光速，但数据无法足够快地跨越芯片以实现更快的处理器速度的增长。

不过虽然存在时钟频率的限制，芯片组件的密度还在持续增加。因此，既然无法提供 6 GHz 的核，芯片厂商转而开始提供双核处理器，每个核运行于 3 GHz。这种趋势还在持续，目前还没有终止的迹象；在 Java 8 发布之时(2014 年 3 月)，四核处理器已经成为主流，八核处理器也已经出现在硬件市场上，而专业服务器早就在每个处理器中放置了几十个核。趋势很明显，不适应这种状况的任何编程模型都会在与适应这种状况的模型的竞争中败下阵来。适应意味着向开发者提供一种可访问的方式，使之能够将多个任务分发到多个核上并行执行，从而利用多核的处

理能力[1]。另一方面，不适应意味着默认情况下 Java 程序会被绑定到单个核上，相对于那些能够帮助用户并行运行代码的语言来说，这会极大地降低程序的运行速度。

本节一开始的代码已经表明了改变的必要性，代码一次只能根据迭代器指定的顺序访问一个列表元素。集合处理并非程序员执行的唯一一个处理器密集的任务，不过它却是最重要的任务之一。Java 循环构建中所用的迭代模型会强制顺序处理集合元素，如果对运行时最迫切的需求刚好相反(至少要考虑性能因素)：将处理分发到多个核上，那么这就会产生严重的问题。第 6 章将会谈到，并不是每个问题都会从并行化中获益，不过最好的情况则是程序的加速几乎与核心数量线性相关。

1.1.1 内部迭代

在考虑将迭代的顺序模型应用到真实世界的场景中时就会发现其侵入性变得非常明显。如果有人通过如下指令让你邮寄一些信件——“重复如下动作：如果还有信件，那么按照收件人姓氏的字母表顺序取出下一个，然后将其放到邮箱中”，那么你可能会觉得他这么说太啰嗦了。你知道在这个任务中顺序并不重要，无论是顺序执行还是并行执行都可以，只不过不要遗漏信件即可。这时，你会觉得在存在更好策略的情况下，外部迭代导致集合只能连续并且按照固定顺序处理元素的做法实在是太低效了。

实际上，对于这个现实任务来说，你只需要知道每封信件都要邮寄出去即可；到底该怎么做取决于你自己。同样，我们应该告诉集合应该对它们所包含的每一个元素采取什么动作，而不是

[1] 将一个处理任务分发到多个处理器上通常称为并行化。即便我们不喜欢这个词，但这个简称还是很有用的，会让我们的解释更加简洁，可读性也更好。

像外部迭代那样指定怎么做。如果能够做到这一点，那么代码会变成什么样子呢？集合只需要公开一个方法来接受“做什么”，也就是说对每一个元素要执行什么任务；这个方法的一个显而易见的名字就是 forEach。借助于它，我们可以像下面这样替换掉本节一开始的迭代代码：

```
pointList.forEach(/*translate the point by (1,1)*/);
```

在 Java 8 之前，这个建议看起来会很奇怪，因为 java.util.List (pointList 的类型)并没有 forEach 方法，作为一个接口，我们也无法向其添加方法。不过，第 7 章将会看到 Java 8 通过引入非抽象接口方法解决了这一问题。

新方法 Collection.forEach(实际上是由 Collection 从其父接口 Iterable 继承的)只不过是内部迭代的一个示例，之所以这么说是因为，虽然已经看不到显式的迭代代码，但迭代依旧在内部发生。迭代现在由 forEach 方法管理，它会对集合中的每个元素应用其行为参数。

从外部迭代到内部迭代的变化看起来很小，只不过是迭代工作跨越了客户端-库的边界。不过，其结果却并不是那么简单。我们所需要的并行工作现在可以定义在集合类中，不必重复写在每一个要迭代集合的客户端方法中。此外，实现上可以自由使用其他技术，比如说延迟加载、乱序执行或是其他方法，从而更快地获得结果。

如果编程模型允许集合库的作者针对每个集合自由选择最佳的批处理实现方式，那么内部迭代就很有必要了。不过要想替换 forEach 调用中的注释，那么该如何告诉集合方法应对每个元素执行什么任务呢？

1.1.2 命令模式

没必要脱离传统的Java机制来寻觅这个问题的答案。比如说，我们会例行公事一般地创建 Runnable 实例并向其传递一些参数。如果你将 Runnable 当作代表了一个任务的对象，当 run 方法调用时就会执行任务，那么你会发现我们现在所需要的与之非常相似。再举一个例子，开发者可以通过 Swing 框架定义一个动作，该动作会执行以响应多种不同的用户界面事件，如菜单项选择、按钮按下等。如果熟悉经典的设计模式，那么你会发现这是对命令模式的描述。

在这种情况下，我们要考虑到底需要什么命令？我们的出发点是调用 List 中每个 Point 的 translate 方法。因此对于该例来说，forEach 应该将一个对象作为参数，该对象会公开一个方法，这个方法会对列表中的每个元素调用 translate。如果该对象是某个更加通用接口的实例，比如说 PointAction，那么就可以对 Point 集合迭代时所要执行的不同行为定义不同的 PointAction 实现。

```
public interface PointAction {
    voiddoForPoint(Point p);
}
```

现在，我们需要的实现如下所示：

```
classTranslateByOne implements PointAction {
    public void doForPoint(Point p) {
        p.translate(1, 1);
    }
}
```

现在可以勾画出 forEach 的简单实现：

```
public class PointArrayList extends ArrayList<Point> {
```

```
    public void forEach(PointAction t) {
        for (Point p : this) {
            5t.doForPoint(p);
        }
    }
}
```

如果让 pointList 成为 PointArrayList 的一个实例，那么就可以通过如下客户端代码实现内部迭代的目标：

```
pointList.forEach(new TranslateByOne());
```

当然，这段玩具代码针对性太强了；我们不应该为需要处理的每一种元素类型都编写一个新的接口。幸好，我们不需要这么做；名字 PointAction 与 doForPoint 没什么特别的；如果通过其他名字替换掉它们，那么结果不会有什么变化。在 Java 8 集合库中，它们称为 Consumer 和 accept。这样，PointAction 接口就变成了下面这样：

```
public interface Consumer<T> {
    void accept(T t);
}
```

参数化接口类型可以让我们省却专门的 ArrayList 子类，并且直接向类本身添加方法 forEach，正如 Java 8 继承所做的那样。该方法接收一个 java.util.function.Consumer，它会接收并处理集合中的每个元素。

```
public class ArrayList<E> {
    ...
    public void forEach(Consumer<E> c) {
        for (E e : this) {
            c.accept(e);
        }
```

```
    }
}
```

将这些改变应用到客户端代码上，结果如下：

```
class TranslateByOne implements Consumer<Point> {
    public void accept(Point p) {
        p.translate(1, 1);
    }
}
...
pointList.forEach(new TranslateByOne());
```

你可能会觉得这段代码还是很笨拙。不过请注意，这种笨拙现在主要体现在类实例每个命令的表示上。在很多情况下，这有些过度了。比如说，在目前的情况下，forEach 真正所需的只是提供给它的对象的单个方法 accept 的行为。之所以包含进构成对象的状态和其他所有信息，是因为在 Java 中，如果不是原生数据类型，那么方法参数必须是对象引用。不过到目前为止，我们总是需要指定这些信息。

1.1.3 lambda 表达式

上一节的最后一段代码并非针对命令模式的惯用 Java 代码。在该例中，如果某个类很小并且不太可能重用，那么更加常见的做法是定义一个匿名内部类：

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

有经验的 Java 开发者对上述代码会很熟悉，以至于我们都忘

记了初次遇到它时的感受。一般来说，初次遇到以这种方式使用的匿名内部类的语法的第一反应就是它太丑陋、冗长，并且难以快速理解，虽然它所做的全部事情只不过是“对每个元素进行该处理”。你不必完全同意这些判断，认为说服开发者使用这种方式来进行每一种集合操作是不可能的事情。我们只不过是通过它来引入 lambda 表达式而已[2]。

要想降低该调用的冗余性，我们应该识别出哪些地方提供的信息“编译器可以从上下文中推断出来”。其中一份信息就是匿名内部类所实现的接口名。编译器完全可以知晓 forEach 的参数类型是 Consumer<T>，足以通过这份信息来检查所提供参数的类型兼容性。我们不再强调编译器可以推断出的代码：

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

其次，被重写的方法名是什么呢，在该例中是 accept 吗？一般来说，编译器是无法推断出此信息的。不过对于 Consumer 来说，无须推断其名字，因为该接口只有唯一一个方法。这种“单方法接口”模式对于定义回调来说是非常有用的，它甚至还有一个官方的表述：用于这种简写形式的任何对象都必须实现一个这样的接口，公开单个抽象方法(这称为函数式接口，有时也称为 SAM 接口)。这样，编译器就可以无歧义地选择出正确的方法。

---

[2] 人们经常好奇于 lambda 这个名字的由来。lambda 表达式的想法来源于美国数学家 Alonzo Church 于 20 世纪 30 年代提出的计算模型，其中希腊字母 λ(lambda)代表着函数抽象。不过为何会选择这个特殊的字母呢？Church 调侃到：问问他自己的选择吧，他的解释提到了排版错误，不过后来他又给出了另一个答案：“12345，上山打老虎”。

下面再次去除可以通过这种方式推断出的代码：

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

最后，Consumer 的实例化类型通常可以从上下文推断出来，在该例中，当 forEach 方法调用 accept 时，它会提供给其一个 pointList 元素，该元素之前被声明为一个 List<Point>。这会将传递给 Consumer 的类型参数标识为 Point，从而让我们可以省略传递给 accept 的参数的显式类型声明。

下面就是去掉 forEach 调用的最后一部分不必要代码之后的内容：

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
      p.translate(1, 1);
    }
});
```

forEach 的参数代表一个对象，它实现了 forEach 所需的接口(Consumer)，这样在对 pointList 元素 p 调用 accept(该接口的唯一一个抽象方法)时，其效果就等价于调用 p.translate(1, 1)。

这里要通过一个额外的语法(“->”)将参数列表与表达式体分隔开来。加上它之后，我们最终得到了一个简单形式的 lambda 表达式。下面就是用于内部迭代的代码：

```
pointList.forEach(p ->p.translate(1, 1));
```

如果从未使用过 lambda 表达式，那么你现在可以将其看作对方法声明的一种简写，将 lambda 参数列表映射为假想的方法的参

数列表，将 lambda 体(通常前面会有一个 return)映射为方法体。下一章将会看到，我们会改变之前示例所用的 lambda 表达式的简单语法，转而使用多个参数和更加复杂的 lambda 体，在这种情况下，编译器将无法推断出参数类型。不过，如果理解了其中的缘由，你就会对使用 lambda 表达式的动机及其所采取的形式有一个基本的认识。

本节已经介绍了很多。总结一下：我们首先考虑了编程模型所需做出的改变以适应变化的硬件架构的要求；然后回顾了集合元素的处理，这使得我们意识到需要有一种简洁的方式来定义待执行的集合的行为；最后，裁剪了匿名内部类定义的额外代码来让我们实现 lambda 表达式的简单语法。

在本章的剩余章节中，我们将会介绍 lambda 表达式实现的一些新的惯用语。我们将会以更加富有表现力的方式实现集合元素的批处理，这些惯用语上的变化会使得库编写者能够更加轻松地利用并行算法，充分发挥新硬件架构的优势，最后再介绍如何通过函数式行为改善 API 的设计。这种微小的改变可以获得巨大的成果！

1.2　从集合到流

我们来扩展一下上一节的示例。在现实的程序中，经常会通过多个步骤来处理集合：我们会迭代处理集合来生成新的集合，后者又会被迭代处理，以此类推。下面来看一个示例，该例是虚构出来的，并且做了一定的简化。该例首先生成了一个 Integer 实例的集合，接下来通过转换生成了一个 Point 实例的集合，最后寻找到距离原点最远的点到原点的距离。

```
List<Integer>intList = Arrays.asList(1, 2, 3, 4, 5);
List<Point>pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i % 3, i / 1));
}
doublemaxDistance = Double.MIN_VALUE;
for (Point p : pointList) {
    maxDistance = Math.max(p.distance(0, 0), maxDistance);
}
```

虽然这段代码还有改进空间，不过这却是大家惯用的 Java 代码——大多数开发者已经看到过很多这种模式的代码——不过如果以新人的眼光来看待它，那么你就会立刻发现一些令人讨厌的特性。首先，代码非常冗长，用了9行代码才实现这3个操作。其次，集合 pointList 只用作中间存储，但却在程序操作中花费了很高的代价；如果中间存储非常大，那么往好的方面说，创建它会增加垃圾回收的成本，往坏的方面说则会导致可用的堆空间消耗殆尽。最后，这里有个难以发现的隐含假设，即空列表的最小值是 Double.MIN_VALUE。不过最糟糕的是开发者的意图与代码所表现出来的意图之间的巨大差异。要想理解该程序，你得搞清楚它做的是什么，然后猜测开发者的意图(如果幸运，还可以阅读注释)，然后通过匹配程序的操作与推理出的非正式规范来检查其正确性[3]。所有这些工作非常耗时间并且极易出错，事实上，高级语言的目标在于通过提供与开发者心智模型非常接近的代码来最小化这其中的工作量。那么该如何减小二者之间的沟壑呢？

下面来重述一遍问题：

“对Integer实例集合中的每一个元素应用变换来生成一个

[3] 现在的情况要比过去好很多了。年龄稍大点的一些人还会记得使用汇编语言(低级语言，与机器码很接近)编写大型程序时的工作量。从那时开始，编程语言的表达力已经丰富了很多，不过还有很大的改进空间。

Point，然后找出距离原点最远的点与原点之间的距离”。

如果去掉上述代码中与该非正式规范不相关的部分，那么我们会发现代码与问题规范之间是非常不匹配的。去掉第一行代码(一开始创建列表 intList 的代码)，我们得到：

```
List<Point>pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i % 3, i / 3));
}
doublemaxDistance = Double.MIN_VALUE;
for (Point p : pointList) {
    maxDistance = Math.max(p.distance(0, 0), maxDistance);
}
```

这展现出了一种新式的、面向数据的程序查看方式，如果了解 Unix 管道与过滤器，那么你就会对这种方式很熟悉：我们可以从源集合开始追寻单个值的处理，看着它首先由一个 Integer 转换为一个 Point，然后由一个 Point 转换为一个 double。这两个转换可以独立进行，无须引用其他被处理的值，这正是并行化的要求。只是对于第 3 步寻找最大距离来说需要值来进行交互(即便如此，我们也有方法来高效地进行并行计算)。

可以通过图形的方式来表达这种面向数据的视图，如图 1-1 所示。在该图中，矩形框代表操作。连接线代表将一个值序列发送到一个操作的新方式。这与任何类型的集合都不同，因为在给定的某个时刻，连接器所传递的值可能尚未产生。这些值序列称为流。流与集合不同，因为它提供了一个可选的有序值序列而无须为这些值提供任何存储；它们是“移动中的数据”，这是一种表示批量数据操作的方式。如果不熟悉流的概念，那么可以想象一种迭代器，其唯一的操作类似于 next，只不过除了返回下一个值之外，它还可以标识出后面已经没有值了。在 Java 8 集合 API 中，

流是通过接口表示的——Stream 代表引用值，IntStream、LongStream 与 DoubleStream 表示原生值的流——其位于包 java.util.stream 中。

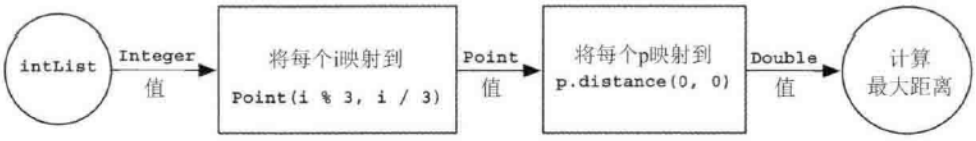


图 1-1　将流组合为数据管道

在这种视角下，图 1-1 中的矩形框所表示的操作都是对流的操作。图中的矩形框表示称为 map 的操作的两种应用；它通过系统规则转换每一个流元素。单独查看 map，我们会认为在操作单个流元素。不过，很快就会看到其他流操作，可以重排列、丢弃，甚至插入值；每个操作都可以描述为接收一个流，然后以某种方式对其执行变换。每个矩形框都代表一个中间操作，它不仅在流上定义，而且还会返回流作为其输出。比如说，假设一个流 intStream 构成了第 1 个操作的输入，那么图 1-1 中间操作所执行的变换就可以通过代码表示为：

```
Stream<Point> points = intStream.map(i -> new Point(
      i % 3, i / 3));
DoubleStream distances = points.mapToDouble(
      p ->p.distance(0, 0));
```

管道最后的圆圈代表终止操作 max。终止操作会消费一个流，并且有可能返回单个值，如果流为空，则什么都不返回，这由一个空的 Optional 或是其专门的原生 Optional 之一来表示：

```
OptionalDouble maxDistance = distances.max();
```

图 1-1 中的管道有开始、中间处理以及结束。我们看到了定义在中间以及结束位置处的操作；那么开始呢？我们可以通过各

种源提供进入流中的值——集合、数组或是生成函数。实际上，常见的情况是将集合的内容提供给流，就像这里所做的那样。Java 8 集合针对该目标提供了一个新方法 stream()，这样管道的开始可以表示为：

```
Stream<Integer> intStream = intList.stream();
```

本节一开始的完整代码就变成了下面这样：

```
OptionalDouble maxDistance =
    intList.stream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p ->p.distance(0, 0))
        .max();
```

这种风格(通常称为流式风格，因为“代码在流动”)并不了解集合处理的上下文，一开始可能很难读懂。不过，相较于本节介绍的代码逐次迭代方式，它很好地平衡了简洁性与对问题声明之间的关系：“将源 intList 中的每个整数映射到相应的 Point 中，将结果列表中的每个 Point 映射到与原点之间的距离，然后找出结果中的最大值”。该代码结构突出了关键操作，而非像一开始那样模糊了这些操作。

作为奖励，创建与管理中间集合所导致的性能消耗也随之消失：顺序执行，流代码要比相应的循环版本快两倍。并行执行对于大型数据集来说会产生非常棒的加速效果。

1.3　从串行到并行

本章一开始就表示 Java 现在需要支持对集合的并行处理，而 lambda 则是提供这种支持的必备一环。我们已经看到客户端代码

通过 lambda 可以多么轻松地使用内部迭代。最后看到了对集合类的内部迭代是如何实现并行处理的。虽然不是每天都会用到，但了解其工作原理还是很有帮助的——对于客户端代码的开发者来说，其实现的复杂性实际上被隐藏了。

在多核上的独立执行是通过为每个核分配一个不同的线程来实现的，每个线程会执行整个工作的一个子任务，在该例中就是待处理的集合元素的一个子集。比如说，假如有一个四核处理器和包含了 N 个元素的一个列表，程序可以定义一个 solve 算法，将任务分解以实现并行执行，就像下面这样：

```
if the task list contains more than N/4 elements {
    leftTask = task.getLeftHalf()
    rightTask = task.getRightHalf()
    doInParallel {
        leftResult = leftTask.solve()
        rightResult = rightTask.solve()
    }
    result = combine(leftResult, rightResult)
} else {
    result = task.solveSequentially()
}
```

上述伪代码非常简化地阐释了并行处理，它使用了名为递归分解的模式——递归地将大任务分解为小任务以并行执行，直到子任务变得“足够小”从而能串行执行为止。实现递归分解要了解如何以这种方式来分割任务，如何执行足够小的任务而不必再进一步分割，如何将这些小任务的执行结果合并到一起。到底该如何分割取决于数据源；在该例中，分割列表有着显而易见的实现。将子任务的执行结果合并起来通常是通过对其应用管道终止操作来实现的；对于本章的示例来说则是要得到两个子任务结果的最大值。

Java 的 fork/join 框架就应用了该模式，只不过它是从线程池中为新的子任务分配线程而不是创建新的线程。显然，重新实现该模式需要编写大量的代码，开发者每次处理一个集合时都得这么做，这并不是我们需要的。这是库应该做的事情！

对于该例来说，库类就是集合；在 Java 8 之前，集合库类可以像这样使用 fork/join 框架，这样客户端开发者就能以并行化的思维来解决业务问题了，这本质就是个性能问题。对于现在这个示例来说，客户端代码只需要做一处变更即可，如下代码所示：

```
OptionalDoublemaxDistance =
    intList.parallelStream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p ->p.distance(0, 0))
        .max();
```

上述代码阐释了 Java 8 中引入并行化的意义所在：明确但不唐突。通过将最初的 Integer 值列表递归分解来实现并行执行，直到子列表变得足够小，然后串行执行整个管道，最后再通过 max 将结果合并起来，如上面的 solve 伪代码所示。到底什么是“足够小”取决于可用核的数量，有时还要考虑列表的性质。图 1-2 展示了如何根据四个核对列表进行分解；在该例中，“足够小”指的是将列表大小除以 4(与之相关的一个问题是确定什么样的列表是“足够大”的，从而值得采取并行执行的方式。第 6 章将会对这个问题进行详细的介绍)。

不唐突的并行化是 Java 8 的一个关键特性；API 的变化赋予了库开发者更大的自由。其中一个重要方面就是库开发者可以挖掘出现代以及未来机器架构所带来的性能改进。

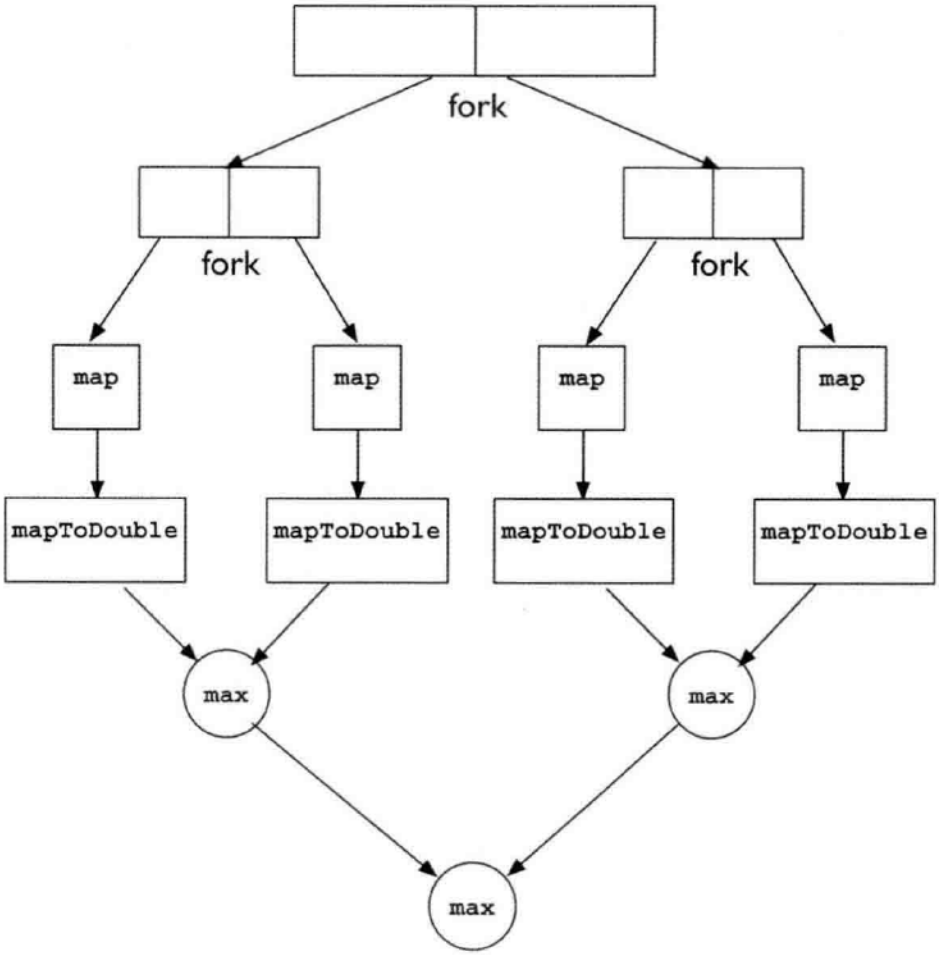


图 1-2　列表的递归分解处理任务

1.4　组合行为

在前面可以看到，从功能上来说，lambda 表达式与匿名内部类十分相似。不过它们二者的写法存在很大的差别，这也导致我们对其思考方式上的不同。lambda 表达式看起来像是函数，因此很自然地我们想知道是否可以让它们的行为像函数那样。这种视角上的转变让我们思考的是对行为而非对象的处理，反过来也会

造成编程方式与库 API 方向上的改变。

比如说，函数的一个核心操作是组合：将两个函数组合起来构成第 3 个函数，其效果等价于连续应用两个组件。组合并非匿名内部类的本性，不过从广义的形态来说，它非常类似于传统的面向对象编程。就像我们会通过解耦来分解面向对象程序一样，组合的反面也适用于函数。

假设我们想要按照 x 坐标对 Point 实例构成的列表进行排序。创建“自定义”排序[4]的标准的 Java 方式是创建一个 Comparator：

```
Comparator<Point> byX = new Comparator<Point>(){
    public int compare(Point p1, Point p2) {
        return Double.compare(p1.getX(), p2.getX());
    }
};
```

就像上一节那样，将匿名内部类声明替换为一个 lambda 表达式会提高代码的可读性：

```
Comparator<Point> byX =
    (p1, p2) ->Double.compare(p1.getX(), p2.getX());
```

❶

不过这么做对于另一个非常关键的问题爱莫能助：Comparator 是呆板的。如果想要定义一个比较 y 坐标而非 x 坐标的 Comparator，那就只能复制整个声明，将声明中的 getX 替换为 getY。好的编程实践促使我们寻求更好的解决方案，短暂的思考过后我们发现 Comparator 实际上执行的是两个函数——从参数中抽取出排序键，然后比较这些键。我们应该可以通过构建一个对

[4] 在 Java 平台中，对象的比较与排序有两种标准方式：类可能会有自然顺序，在这种情况下，它会实现接口 Comparable，这样就会公开出一个 compareTo 方法，对象可以通过它将自身与其他对象进行比较。此外，还可以创建一个 Comparator，就像该例一样。

这两个组件参数化的 Comparator 函数来改进代码❶，现在就来做。这个中间过程看起来既笨拙又冗长，但请坚持下去：最后的结果还是值得我们这么做的。

首先，将已有的两个具体的组件功能转换为 lambda 形式。我们知道关键字抽取函数的函数接口类型(Comparator)，不过还需要知道与函数 p -> p.getX()相对应的函数接口类型。查看专门为函数接口声明设计的包 java.util.function，我们会发现接口 Function：

```
public interface Function<T,R> {
   public R apply(T t);
}
```

现在就可以为关键字抽取与关键字比较编写lambda表达式了：

```
Function<Point,Double> keyExtractor = p ->p.getX();
Comparator<Double> keyComparer = (d1, d2) ->Double.compare(d1,
d2);
```

可以通过如下两个小函数来重新装配新版的 Comparator<Point>：

```
Comparator<Point> compareByX = (p1, p2) -> keyComparer.
compare(keyExtractor.apply(p1), keyExtractor.apply(p2));
```

❷

这与❶的形式相匹配，不过却是一个重要的改进(这个改进在更复杂的情形下体现得更为明显)：你可以插入事先定义好的任何 keyComparer 或 keyExtractor。毕竟，我们的目标就是对构成更大的函数的小组件进行参数化。

虽然按照这种方式重新构建的 Comparator 改进了结构，但却失去了❶的简洁性。我们可以将其具体化，不过更为常见的情形则是 keyComparer 表达了对抽取的键的自然排序。那么❷可以重写为：

```
Comparator<Point>compareByX = (p1, p2) ->
keyExtractor.apply(p1).compareTo(keyExtractor.apply(p2));
```
❸

注意到该特例的重要性，平台库的设计者向接口 Comparator 添加了一个静态方法 comparing：给定一个关键字抽取器，它会使用针对键的自然顺序创建一个相应的 Comparator[5]。下面是其方法声明，其中的泛型类型参数已经进行了简化：

```
public static <T,U extends Comparable<U>>
          Comparator<T>comparing(Function<T,U>keyExtractor) {
  return (c1, c2) ->
    keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

相对于❸，我们可以通过该方法编写如下代码(假设已经添加了对 Comparators.comparing 的静态导入声明)：

```
Comparator<Point> compareByX = comparing(p ->p.getX());
```
❹

相对于❶来说，❹是一个重大的改进：更加简洁，更容易理解，这是因为它隔离并且强调了重要的元素(关键字抽取器)，之所以可以这样是因为 comparing 接受了一个简单的行为，并且通过它构建出更为复杂的行为。

为了能立刻看到改进，假设问题发生了一些变化，相对于找到距离原点最远的点，我们决定按照它们与原点之间的距离的升序将所有点打印出来。得到这种排序是轻而易举的事情：

```
Comparator<Point> byDistance = comparing(p ->p.distance(0, 0));
```

要想实现改变后的问题规范，我们只需要对流管道做小小的修改即可：

---

[5] 还能以相同的方式为原生类型创建 Comparator，不过由于无法使用自然排序，因此它们会使用封装类提供的 compare 方法。

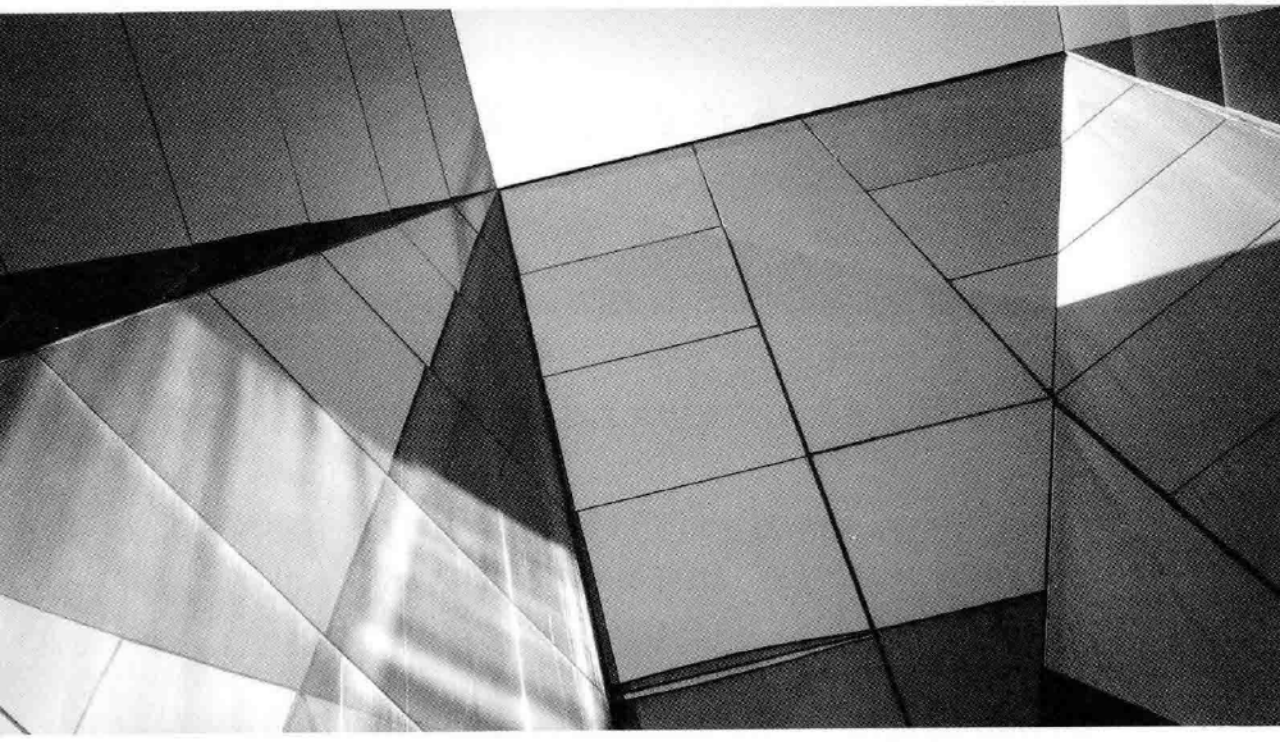
```
    intList.stream()
        .map(i -> new Point(i % 3, i / 3))
        .sorted(comparing(p ->p.distance(0, 0)))
        .forEach(p ->System.out.printf("(%f, %f)", p.getX(),
p.getY()));
```

需要适应新的问题域的改变表明了 lambda 所带来的一些好处。改变 Comparator 是非常直接的，因为它是通过组合创建的，我们只需要指定改变的单个组件即可。新的比较器非常平滑地适合了现有的流操作，新的代码再一次非常接近于新的问题域，问题与代码之间变化的部分天然合一。

1.5 小结

现在，大家应该很清楚为何说 lambda 表达式是万众期待的了。在本章刚开始的小节中，我们看到了它们是如何提升性能的，库开发者可以通过 lambda 表达式实现自动的并行化。虽然这种改进未必是普遍的——本书的一个目标就是帮助你理解应用何时会从“并行化”中获益——但它代表了向正确方向前进的一大步，让应用程序员能够利用现代硬件的性能改进。

最后一节介绍了lambda是如何促使我们编写出更棒的API的。Comparator.comparing的签名就是一个信号：随着客户端程序员逐步习惯于提供诸如comparing所接受的关键字抽取函数，comparing这样细粒度的库方法将会成为标准，借助于此，客户端编码的风格将会得到改进，代码也会得到简化。



第 2 章

Java lambda 表达式的基础知识

第 1 章简要介绍了 lambda 表达式以及将其引入 Java 中的动机。本章将会详细介绍 lambda 表达式的定义、如何将其应用于 Java 程序中，以及在什么地方应用。

2.1 lambda 表达式的定义

一般来说，在数学与计算中，lambda 表达式指的是一个函数：对于输入值的部分或全部组合来说，它会指定一个输出值。到目前为止，在 Java 中还没有办法编写独立的函数。我们常常使用方法来代替函数，不过它总是作为对象或类的一部分而存在。现在，lambda 表达式提供了一种与独立函数更为近似的方式[1]。在传统的 Java 术语中，lambda 可以看成一种匿名方法，拥有更为简洁的语法，可以省略修饰符、返回类型、throws 语句，在某些情况下还可以省略参数类型。

lambda 的语法

Java 中的 lambda 表达式包含一个参数列表和一个 lambda 体，二者之间通过一个函数箭头“->”分隔。第 1 章的示例都包含了单个参数：

```
p ->p.translate(1, 1)
i -> new Point(i, i + 1)
```

不过与方法声明类似，lambda 可以接收任意数量的参数。除了像之前那样接收单个参数的 lambda 外，参数列表必须使用圆括号包围起来：

```
(x, y) -> x + y
() -> 23
```

[1] 关于 lambda 的一个常见问题就是它们是否是传统意义上定义在 Java 中的对象。这个问题并没有简单的答案，因为虽然 lambda 表达式会计算对象引用，但它们在其他方面的行为与对象则是完全不同的，请参见 2.2.1 节。

到目前为止，我们在声明参数时并没有显式指定类型，因为不指定类型时 lambda 的可读性通常会更好一些。不过，我们总是可以提供参数类型，有时这也是必要的，因为编译器可能无法从上下文中推断出其类型。如果显式提供了类型，那就必须为所有参数都提供类型，而且参数列表必须包围在圆括号中：

```
(int x, int y) -> x + y
```

可以像方法参数那样修改这种显式类型的参数，例如，可以将其声明为 final，也可以添加注解。

函数箭头右侧的 lambda 体可以是表达式，到目前为止所有示例都是这样的(注意，方法调用是表达式，包括那些返回 void 的方法)。诸如此类的 lambda 有时也称为“表达式 lambda”。更为一般的形式则是“语句 lambda”，其中的 lambda 体是一个块，也就是说，是由花括号包围的一系列语句：

```
(Thread t) ->{ t.start(); }
() ->{ System.gc(); return 0; }
```

表达式 lambda：

```
args -> expr
```

可以看成相应的语句 lambda 的简写形式：

```
args -> { return expr; }
```

在块体中到底使用还是省略 return 关键字的原则与普通的方法体是一致的，也就是说，如果 lambda 体中的表达式有返回值，那就需要使用 return，也可以后跟一个参数来立刻终止 lambda 体的执行。如果 lambda 返回 void，那就可以省略 return，也可以使用它，但后面不带参数。

lambda表达式不需要也不允许使用throws语句来声明它们可能会抛出的异常。

2.2 lambda与匿名内部类

如果看过了第1章由匿名内部类到lambda表达式的转换，那么你可能想知道，除了具体语法外，二者之间真正的差别是什么。事实上，lambda表达式有时被错误地称为匿名内部类的“语法糖”，这说的是二者之间只存在简单的语法上的变化。但实际上，二者之间存在很多显著差异，其中有两点对于程序员来说非常重要：

- 内部类创建表达式会确保创建一个拥有唯一标识的新对象，而lambda表达式的计算结果可能有，也可能没有唯一标识，这取决于具体实现。相对于对应的内部类来说，这种灵活性可以让平台使用更为高效的实现策略。
- 内部类的声明会创建出一个新的命名作用域，在这个作用域中，this与super指的是内部类本身的当前实例；相反，lambda表达式并不会引入任何新的命名环境。这样就避免了内部类名称查找的复杂性，名称查找会导致很多小错误，例如想要调用外围实例的方法时却错误地调用了内部类实例的Object方法。

接下来的两节将会对这两点进行介绍。

2.2.1 无标识性问题

到目前为止，Java程序的行为总是与对象相关联，以标识、状态和行为为特征。lambda则违背了该规则；虽然它们会共享对象的一些属性，但其唯一的用处是表示行为。由于没有状态，因

此标识问题就不重要了。语言规范显式表示其是未确定的，唯一的要求就是 lambda 必须计算出实现了恰当函数接口(参见 2.4 节)的类实例。这么做的意图是赋予平台足够的灵活性来进行优化，如果每个 lambda 表达式都要拥有唯一标识，那么这种灵活性无法实现。

2.2.2　lambda 的作用域规则

就像大多数内部类一样，匿名内部类的作用域规则非常复杂，这是因为它可以引用从父类型继承下来的名字，以及声明在外部类中的名字。lambda 表达式则要简单得多，因为它们并不会从父类型中继承名字[2]。除了参数以外，用在 lambda 表达式体中的名字的含义与体外面是一样的。例如，像下面这样在 lambda 中再次声明一个局部变量就是非法的：

```
    void foo() { final int i = 2; Runnable r = () -> { int i = 3;}}
// illegal
```

参数就像局部声明一样，因为它们可以引入新的名称：

```
IntUnaryOperator iuo = i ->{ int j = 3; return i + j; };
```

lambda 参数与 lambda 体局部声明可以隐藏字段名(也就是说，字段名可能会临时被重新声明为参数或局部变量名)。

```
class Foo {
   Object i, j;
   IntUnaryOperator iuo = i -> { int j = 3; return i + j; }
}
```

---

[2] 这个规则会将父类型(也就是函数接口)中声明的任何名字排除在lambda作用域之外。除了抽象方法外，接口可以声明静态的final字段、静态嵌套类以及默认方法(参见第7章)。它们都不在实现的lambda的作用域之内。

由于 lambda 声明就像简单的块一样，因此关键字 this 与 super 与外围环境的含义一样：也就是说，它们分别指的是外围对象及其父类对象。例如，如下程序会向控制台打印出两条“Hello, world!”消息：

```
public class Hello {
  Runnable r1 = () -> { System.out.println(this); };
  Runnable r2 = () -> { System.out.println(toString()); };

  public String toString() { return "Hello, world!"; }

  public static void main(String... args) {
     new Hello().r1.run();
     new Hello().r2.run();
  }
}
```

如果使用匿名内部类而非 lambda 表达式来编写同样的程序，那么它会打印出在内部类的对象上调用 toString 方法的结果。对于匿名内部类来说，更为常见的访问外围对象当前实例的用法要使用笨拙的语法 OuterClass.this，而这对于 lambda 来说是非常直接的。

关于如何解释 this：常常会有这样一个问题：lambda 能否引用自身呢？如果名字在作用域中，那么 lambda 就可以引用自身，不过初始化器中的前向引用限制规则(对于局部变量与实例变量均如此)导致 lambda 变量无法初始化。我们还是可以声明一个递归定义的 lambda：

```
public class Factorial {
   IntUnaryOperator fact;
   public Factorial() {
      fact = i -> i == 0 ? 1 : i * fact.applyAsInt(i - 1);
   }
}
```

需要递归的 lambda 定义的场合并不太多，这种做法完全可以胜任该任务。

2.3　变量捕获

上一节介绍了如何对 lambda 表达式从其外围环境中继承下来的名字进行解释。不过名字解释只是一部分；一旦理解了从环境中继承下来的变量名的含义，我们还需要知道能对它们做什么，应该做什么，这是两件不同的事情。

首先，我们发现很多实用的 lambda 表达式实际上并不会从其环境中继承任何名字。面向对象程序员可以通过与静态方法做类比来理解这一点；虽然一般来说，对象的行为依赖于其状态，但很多时候我们定义的方法并不会依赖于系统状态。例如，辅助类 java.lang.Math 只包含了静态方法，在计算数字的平方根时考虑系统状态是毫无意义的事情。lambda 可以承担同样的角色，与调用 Math.sqrt 得到相同结果的 lambda 如以下代码所示：

```
DoubleUnaryOperator sqrt = x -> Math.sqrt(x)
```

像这种只通过参数和返回值与环境进行交互的 lambda 称为无状态或非捕获 lambda。与之相反，捕获 lambda 会访问外围对象的状态。“捕获”是个技术术语，指的是保留住 lambda 对其环境的引用。其含义表示变量已经被 lambda 引用到，可以对其进行查询；在其他语言中，后面在计算该 lambda 时还可以修改该变量。

捕获所提供的访问能力是有限制的；这种限制的中心原则就是捕获变量的值不能修改。因此，虽然传统叫法是“变量捕获”，不过事实上称为“值捕获”更为准确一些。为了理解该原则是如

何实现的，我们首先来考虑局部变量捕获；接下来再看看字段捕获，这样就更好理解了。

传统上，一般对于局部类，特别是匿名内部类来说，要想在内部类中访问到外部方法中的局部变量，这些局部变量需要声明为 final。该规则也非常类似于 Java 8 中的 lambda，只不过语法上的要求没那么严格而已：要想确保 lambda 不会修改从外部环境中捕获到的变量值，该变量必须定义为 final，这意味着变量在初始化后不会在任何地方被赋值(从 Java 8 开始，匿名类与局部类也可以访问到 final 变量)。

本质上，对于那些被当作 final 的变量来说，我们可以省略声明中的 final 关键字。相比于其他语言，Java 中只能对 final 变量进行捕获的限制引起了很多争论，例如 JavaScript 就没有这个限制。防止 lambda 修改局部变量的理由是这么做会引入非常复杂的变化，而这些变化会影响到程序的正确性与性能，无论如何，这么做都是毫无必要的：

- **正确性**：放开这个限制会引入一类与局部变量相关的多线程Bug。在Java中，到目前为止局部变量都不会产生竞态条件与可见性问题，这是因为它们只能由执行局部变量所在方法的线程访问到。不过我们可以将lambda从创建它的线程传递给其他线程，这样如果第2个线程中的lambda可以修改局部变量，那么方才提到的竞态条件与可见性问题就会出现。

  此外，无论涉及多少个线程，lambda的生命周期可能会大于对计算它的方法的调用。如果捕获的局部变量是可变的，那么其生命周期就得比创建它们的方法调用长。抛开其他后果不谈，这种改变会引入与局部变量相关的内存泄漏问题。

- **性能**：如果对变量的访问是同步保护的，那么允许多线程访问可变变量的程序的正确性就是可以保证的。不过这么做的代价却不太符合引入 lambda 的一个主要目标——将对不同参数的函数的计算分发到不同线程上这一策略。甚至从不同线程中读取可变的局部变量的值都得引入同步或是使用 volatile，从而避免读取到旧数据。
- **非根本性**：审视该限制的另一种方式就是考虑它不允许使用的场景。该限制禁止使用的惯用法涉及初始化与变化，就像下面这个对一个 List<Integer>中的值求和的示例：

```
int sum = 0;
integerList.forEach(e -> { sum += e; }); // illegal
```

Stream API 提供了更棒的选择。对于这个简单的示例来说，我们可以写成：

```
int sum = integerList.stream()
        .mapToInt(Integer::intValue)
        .sum();
```

后续章节将会详细介绍，Java 8 设计的一个指导原则就是学习这种函数式风格要比其所带来的代码质量上的改进更为重要。这种风格的代码也是面向并行的(参见 3.1.1 节)，这可以看作一个额外的好处，在目前的某些情况下它会带来更棒的性能，未来这种情况将会更多。

其实对 final 的限制很容易就能规避，这也不是什么秘密了。例如，如果局部变量是个数组引用，变量是 final 的，不过数组内容还是可变的。可以通过这个众所周知的技巧实现迭代处理，不过在并行执行时，你很可能无意间就造成了竞态条件。可以通过同步来防止竞态条件的出现，不过这么做会导致竞争加剧并降低

性能[3]。一言以蔽之，不要这么做！

看起来对字段名的捕获没有诸如 final 之类的限制，但其实际上与局部变量是一样的：对字段名 foo 的引用实际上是 this.foo 的简写，其中的伪变量 this 担负着不可变的局部变量的角色。正如对任何其他对象字段的引用一样，如果 lambda 就是对象引用，那么在该示例中所捕获到的值就是 this。在计算 lambda 时，this 会被解引用(就像对任何局部引用变量一样)，字段会被访问。

似乎允许修改局部变量的呼声也应该出现在字段上。不过 lambda 打算提供一种更加温和的方式，有更好的实现方式而不是将 Java 彻底转变成函数式语言。在 Java 8 之前，修改共享变量是很轻松的事情，非常容易！开发者的职责就是避免这么做，如果可能，还要将其管理起来。通过 lambda 来修改字段值也没有改变这一状况。

2.4 函数式接口

从 1.1.3 节我们知道，lambda 表达式必须实现一个函数式接口。不过，要想实际使用 lambda 与函数式接口，我们需要更准确地理解二者之间的关系。本节将会概览并简要介绍库 java.util.function 所提供的函数式接口。稍后(2.7 节)将会更详尽地介绍二者之间的关系。

函数式接口之所以起到这种核心作用源自于其最为重要的一个属性，这个属性赋予了函数式接口的名字：它们可用于描述函数的类型。例如，接口 UnaryOperator：

---

[3] 更安全的做法也是存在的，例如 AtomicInteger 或 LongAdder。不过更好的做法则是在尽可能的情况下不要同时修改共享变量。

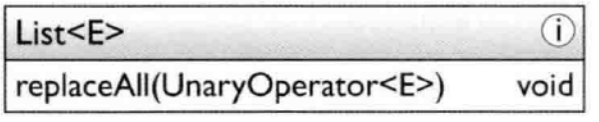
```
public interface UnaryOperator<T> { T apply(T t); }
```

描述了如下函数：

```
f: T -> T
```

这是其函数类型，也是最为简单、最为常见的情况，仅仅是函数式接口的单个抽象方法的方法类型：即方法的类型参数、形参类型、返回类型，以及在适用情况下的抛出类型(函数类型之前称为“函数说明符”；你可能还会遇到这个术语)。2.7.1 节将会更为详尽地介绍函数类型。

函数类型是 lambda 必须匹配的，这包括通过装箱或拆箱、增大或缩小范围等手段实现的类型适配。例如，假设我们将变量 pointList 声明为 List<Point>，现在想要替换里面的每一个元素。方法 replaceAll 就可以很好地实现这个目标：



类图约定

本书中的平台库 API 的类图采用了一些简化的缩写：

- 右上角的图标包含了“i”或“s”，表示类图是否包含实例方法或静态方法。
- 省略泛型参数类型的通配符边界(例如，forEach 的参数实际上是 Consumer<? extends T>)。
- 如果发现方法声明中的类型变量没有出现在类声明中，那就可以假定它是方法的类型参数。
- 类图不一定是完整的，只会列出对于讨论来说重要的方法。

调用可以像下面这样：

```
pointList.replaceAll((Point p) -> { /* return new Point object
*/ });
```

要想编译上述代码，lambda表达式需要匹配UnaryOperator<Point>的函数类型，它是如下方法的类型：

```
public Point apply(Point p);
```

该例直观易懂，不过一般来说，匹配过程会对类型检查提出一些挑战。之前，任何结构良好的 Java 表达式都会有一个定义类型；现在，虽然类型良好的语句或表达式中的每个 lambda 都会实现一个函数式接口，但 lambda 本身只会部分决定它到底实现了哪个函数式接口。上下文需要提供足够多的信息才能够推断出确切的类型，这个过程称为目标类型[4]。下面这个示例展示的 lambda 表达式就拥有多种可能的类型：

```
x -> x * 2
```

该表达式可以是多种函数式接口的实例，包括声明在 java.util.function 中的下面两个。

```
public interface IntUnaryOperator {
   int applyAsInt(int operand);
}
public interface DoubleUnaryOperator {
```

[4] 实际上，目标类型并不是全新的概念，不过它在 Java 8 中广泛用于编译 lambda 表达式，这与传统 Java 程序的类型检查是不同的。现在，表达式的类型会先被计算出来，然后再根据其上下文进行兼容性检查；如果上下文是一个方法调用，那么恰当的重载就会作为上下文。与之相反，无类型的 lambda 表达式并没有要与上下文比对的类型，这样在方法调用的情况下，首先就要选择重载，不过依然要与 lambda 保持兼容。2.7 与 2.8 节将会介绍对编译器提出的这种挑战。

```
    double applyAsDouble(double operand);
}
```

这样，如下两个赋值：

```
IntUnaryOperator iuo = x -> x * 2;
DoubleUnaryOperator duo = x -> x * 2;
```

都是合法的，因为每种情况下 lambda 表达式的类型都与目标类型兼容，也就是被赋值变量的函数式接口类型。编译器通过将 lambda 参数当作 int 类型来确定第一种情况的类型，这样 lambda 整体就是从 int 到 int 的函数。这与 IntUnaryOperator 的函数类型匹配，它也接收一个 int 并返回一个 int。与之类似，在第二种情况下，lambda 参数可以当作 double 类型，这样 lambda 就是从 double 到 double 的函数了，它与 DoubleUnaryOperator 的函数类型相匹配。

这两个 lambda 表达式从字面上看是一样的，就像我们之前看到的那样，不过它们的类型不同，含义也不同；对 int 值的操作与对 double 值的操作是不同的。类型是在编译期建立起来的，并且不能改变，因此我们无法为不同的类型重用相同的 lambda 表达式(除非它们可以通过类型转换保持兼容)，下一节将会介绍相关示例。

表 2-1 展示了 java.util.function 中 4 种基本的函数式接口类型，并且给出了示例用例以及 lambda 实例(其中的类型参数 T 被实例化为 String，U 被实例化为 Integer)。

java.util.function 中定义的 40 多个奇怪的类型都是通过 3 种不同路由的各种组合由这 4 个类型演化而来的：

表 2-1 java.util.function 中基本的函数式接口

接　口	参数类型	返回类型	示 例 用 例	示　例
Consumer<T>	T	void	使用对象过滤值工厂方法进行转换或是从对象中选择	s ->System.out.print(s)
Predicate<T>	T	boolean		s ->s.isEmpty()
Supplier<T>	无	T		() -> new String()
Function<T,U>	T	U		s -> new Integer(s)

- 原生特化：这些接口使用原生类型替换掉类型参数。代码如下：

```
interface LongFunction<R> { R apply(long value); }
interface ToIntFunction<T> { intapplyAsInt(T value); }
interface LongToIntFunction { intapplyAsInt(long value); }
```

- Consumer、Predicate 与 Function 的函数类型都接收单个参数。有接收两个参数的相应接口，代码如下：

```
interface BiConsumer<T,U> { void accept(T t, U u); }
interface BiFunction<T,U,R> { R apply(T t,U u); }
interface ToIntBiFunction<T,U> { int apply(T t, U u); }
```

- Function 的常见用法要求其参数与结果拥有相同的类型。之前在 List.replaceAll 的参数中已经看到了这一点。我们可以通过将 Function 的变种特化为相应的 Operator 来达成所愿，如下所示：

```
interface UnaryOperator<T> extends Function<T,T> { ... }
interface BinaryOperator<T> extends BiFunction<T,T,T> { ... }
interface IntBinaryOperator { int applyAsInt(int left, int
right); }
```

这个库是“入门套件”，旨在涵盖函数式接口的常见用例。如果你的用例没有涵盖，那么可以很轻松地声明自己的函数式接口，不过最佳实践则是在可能的情况下使用库中的函数式接口。此外，使用@FunctionalInterface 来注解自定义的函数式接口声明也是最佳实践，这样编译器就可以检查接口只会声明一个抽象方法，同时相应的 Javadoc 也会自动添加说明部分。

2.5　使用 lambda 表达式

当对函数式接口实例的引用有效时就可以使用 lambda 表达式，前提是上下文提供了恰当的目标类型，也就是说，上下文明确需要一个函数式接口类型。例如，如下声明的目标类型：

```
IntPredicate ip = i -> i > 0;
```

是 IntPredicate，这是一个函数式接口，其函数类型兼容于 lambda 表达式 i -> i > 0。相反，声明：

```
Object o = i -> i > 0; // invalid
```

就无法通过编译，因为上下文所需的目标类型并不是函数式接口类型，因此它无法给编译器提供编译该 lambda 时所需的类型信息。

有 6 种上下文可以提供恰当的目标类型：

- 方法或构造器参数，在这种情况下，目标类型就是恰当的参数类型。第 1 章已经介绍过这类示例。
- 变量声明与赋值，在这种情况下，目标类型就是被赋值的类型。

```
Comparator<String> cc =
   (String s1, String s2) -> s1.compareToIgnoreCase(s2);
```

数组初始化器与之类似，只不过其目标类型是数组组件的类型：

```
IntBinaryOperator[] calculatorOps = new IntBinaryOperator[]{
  (x,y) -> x + y, (x,y) -> x - y, (x,y) -> x * y, (x,y) -> x / y
};
```

lambda 数组应用范围有限，不过，由于大多数函数式接口都是泛型的，因此泛型数组的创建是不允许的。

- 返回语句，在这种情况下，目标类型是方法的返回类型：

```
Runnable returnDatePrinter() {
   return () ->System.out.print(new Date());
}
```

- lambda 表达式体，在这种情况下，目标类型是 lambda 表达式体所期望的类型，它由外层目标类型推导而来。考虑如下代码：

```
Callable<Runnable> c = () -> () -> System.out.println("hi");
```

这里的外层目标类型是 Callable<Runnable>，其函数类型就是如下方法的类型：

```
Runnable call() throws Exception;
```

因此，lambda 体的目标类型就是 Runnable 的函数类型，即 run 方法的类型。它不接收参数，不返回值，因此与内部的 lambda 匹配。

- 三元条件表达式，在这种情况下，两边的目标类型都由上下文提供。例如：

```
Callable<Integer> c = flag ? (() -> 23) : (() -> 42);
```

- 类型转换表达式，它会显式提供目标类型。例如：

```
Object o = () -> "hi";              // illegal
Object s = (Supplier) () -> "hi";
Object c = (Callable) () -> "hi";
```

第 1 个声明是不合法的，因为没有合适的目标类型能够解决含义模糊的 lambda 表达式声明。第 2 个与第 3 个是合法的，因为类型转换提供了目标类型。该例还表明字面上相同的 lambda 可能会有不同的类型；像下面这样尝试通过不同的类型重用 lambda：

```
Callable c1 = (Callable) s;
```

可以编译通过，不过在运行时会抛出 ClassCastException 异常。

2.6　方法与构造器引用

我们已经知道，一般来说，任何 lambda 表达式都可以看作声明在函数式接口中的单个抽象方法的实现。不过，当 lambda 表达式只是调用现有类中的具名方法的一种方式时，编写 lambda 的更好方式则是使用已有的名字。例如，考虑如下代码，它会向控制台输出列表中的每个元素：

```
pointList.forEach(s -> System.out.print(s));
```

这里的 lambda 表达式只是将参数传递给 print 调用。诸如此类的 lambda(其唯一目的就是将参数提供给一个具体方法)完全是由该方法类型定义的。因此，假如可以通过某种方式确定出类型，那么只包含方法名的简短形式所提供的信息就与完整的 lambda

表达式一样，但可读性会更好。相比于上述代码，我们可以这样编写：

```
pointList.forEach(System.out::print);
```

它表示相同的含义。这种对现有类的具体方法的操作写法称为方法引用。有 4 种类型的方法引用，如表 2-2 所示。本节后续部分将会介绍每种类型的目的。

表 2-2 方法引用类型

名　字	语　法	相应的 lambda 表达式
静态	RefType::staticMethod	(args) -> RefType.staticMethod(args)
绑定实例	expr::instMethod	(args) -> expr.instMethod(args)
未绑定实例	RefType::instMethod	(arg0,rest) -> arg0.instMethod(rest)
构造器	ClsName::new	(args) -> new ClsName(args)

2.6.1 静态方法引用

静态方法引用的语法只需要类与静态方法名，中间通过两个冒号分隔。例如：

```
String::valueOf
Integer::compare
```

是对静态方法的引用。为了理解如何使用静态方法引用，假设我们想要根据大小对一个 Integer 数组排序，数组中的每个值都被看成无符号的。Integer 的自然顺序根据数字来排序(也就是考虑到值的符号)，因此我们需要提供一个显式的 Comparator。我们可以使用静态方法 Integer.compareUnsigned：

```
(x,y) -> Integer.compareUnsigned(x, y);
```

这样，对数组 integerArray 排序就可以调用：

```
Arrays.sort(integerArray, (x,y) -> Integer.compareUnsigned(x,
y));
```

这是合法的，不过这么做要比相应的静态方法引用冗长和重复：

```
Arrays.sort(integerArray, Integer::compareUnsigned);
```

事实上，该方法是在 Java 8 中引入的，并且就是期望按照这种方式使用。未来，API 设计的一个要素就是希望方法签名要适合于函数式接口转换。

注意表 2-2 中的语法 ReferenceType::Identifier 并不总是代表对静态方法的一个引用。后面将会看到，该语法还可用于引用实例方法。

2.6.2　实例方法引用

有两种方式可以引用实例方法。绑定方法引用类似于静态引用，只不过是通过 ObjectReference::Identifier 替换 ReferenceType::Identifier。之前的示例就是绑定方法引用：forEach 方法用于将集合中的每个元素传递给 PrintStream 对象 System.out 的实例方法 print 进行处理，如下 lambda 表达式：

```
pointList.forEach(p ->System.out.print(p));
```

可以替换为绑定方法引用：

```
pointList.forEach(System.out::print);
```

之所以称为绑定引用，是因为接收者已经确定为方法引用的

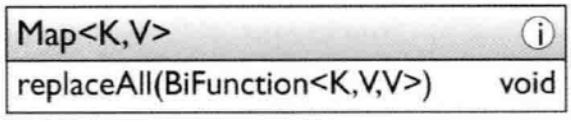
一部分。对方法引用 System.out::print 的每次调用都会有相同的接收者：System.out。不过，你常常会在调用方法引用时带上方法接收者及其参数(来自于方法引用的参数)。要想做到这一点，你需要一个未绑定的方法引用，之所以起这个名字，是因为接收者是不确定的；方法引用的第一个参数被用作接收者。在只有一个参数的情况下，未绑定方法引用是最容易理解的；例如，要想通过工厂方法comparing创建一个Comparator，我们可以将如下lambda表达式：

```
    Comparator personComp = Comparator.comparing(p ->
p.getLastName());
```

替换为未绑定方法引用：

```
    Comparator personComp =
Comparator.comparing(Person::getLastName);
```

未绑定方法引用可以通过其语法识别出来：与静态方法引用一样，我们也使用格式 ReferenceType::Identifier，不过这里的 Identifier 指的是实例方法而非静态方法。要想探寻绑定与未绑定方法引用之间的差别，考虑调用方法 Map.replaceAll 并提供绑定与未绑定的实例方法引用：



Map.replaceAll的效果是对map中的每个键值对应用其BiFunction参数，并使用结果替换键值对中的值部分。如果变量map指向的是一个TreeMap，并且其字符串表示如下：

```
{alpha=X, bravo=Y, charlie=Z}
```

那么像下面这样通过绑定方法引用来调用 replaceAll：

```
String str = "alpha-bravo-charlie";
map.replaceAll(str::replace)
```

效果就相当于三次应用 str.replace，即：

```
str.replace("alpha","X")
str.replace("bravo","Y")
str.replace("charlie","Z")
```

每次调用的结果都会替换相应的值，执行完毕后 map 将包含：

```
{alpha=X-bravo-charlie, bravo=alpha-Y-charlie,
charlie=alpha-bravo-Z}
```

现在使用 map 的初始值来重新执行该例，再次调用 replaceAll，这次使用未绑定方法引用 String::concat，这是对 String 实例方法的引用，接收单个参数。使用单个参数的实例方法作为 BiFunction 看起来有些奇怪，不过事实上它是 BiFunction 的方法引用：它会传递两个参数(键值对)并将第一个参数作为接收者，因此方法本身会像下面这样调用：

```
key.concat(value)
```

方法引用的第 1 个参数移到了接收者的位置，第 2 个参数(以及随后的参数，如果存在的话)会向左移动一个位置。因此，如下调用：

```
map.replaceAll(String::concat)
```

的结果是：

```
{alpha=alphaX, bravo=bravoY, charlie=charlieZ}
```

2.6.3 构造器引用

方法引用是对现有方法的句柄，与之类似，构造器引用是对现有构造器的句柄。构造器引用的创建语法类似于方法引用，只不过使用关键字 new 替换方法名。例如：

```
ArrayList::new
File::new
```

与方法引用一样，对于重载构造器的选择是通过上下文的目标类型实现的。例如，在如下代码中，map 参数的目标类型是类型为 String -> File 的函数；为了与之匹配，编译器会选择带有单个 String 参数的 File 构造器。

```
    Stream<String> stringStream = Stream.of("a.txt", "b.txt",
"c.txt");
    Stream<File> fileStream = stringStream.map(File::new);
```

2.7 类型检查

本节将会介绍 lambda 与某个函数类型匹配所需的条件，下一节将会介绍类型推断是如何在多个重载方法中找到与调用“最为匹配”的那个方法的。这两节内容都很详尽，你可以考虑在第一次阅读时将其跳过或是快速浏览一遍。

在本章前面我们看到如果 lambda 表达式所处的上下文提供了一个目标类型，那么我们就可以说该 lambda 表达式拥有单个类型。诸如此类的表达式(类型在一定程度上是由上下文决定的)称为聚合表达式(poly expressions)。方法与构造器引用也是聚合表达式；如果缺少上下文，那么表达式 String::valueOf 就可以指向名字为 valueOf 的 9 种不同的 String 方法。本节与下节将会探究

lambda 与方法及构造器引用类型检查的过程。不过，由于类型检查针对的是函数式接口的函数类型，因此首先需要对函数类型进行精确定义。

2.7.1　何为函数类型

在本章的后续部分，函数类型的概念对于理解类型检查的工作方式来说是非常重要的，因此我们需要好好地理解它。事实上，2.4 节给出的简单定义在大多数情况下都足够了：函数式接口的函数类型就是其唯一一个抽象方法的类型，也就是其类型参数，再加上参数类型、返回类型与抛出异常类型。

不过，有两个因素会导致这个简单定义变得复杂：首先，所有接口都声明了(如果没有显式声明，那就会隐式声明)与 Object 中的公共方法相对应的抽象方法，函数式接口的单个抽象方法并不在其中。很多时候，Object 方法都是隐式声明的，但并非总是这样；例如，Comparator 就显式声明了 equals 方法，它与 Object 的 equals 方法相匹配。在这种情况下，显式声明不会产生任何影响；Comparator 依然要满足函数式接口的定义。

第 2 个复杂因素就是两个接口可能包含不同的方法，但由于类型擦除的原因导致这两个方法出现了关联。例如，如下两个接口的方法：

```
interface Foo1 { void bar(List<String>arg);}
interface Foo2 { void bar(List arg);}
```

就是重写相关的。如果一个接口的父接口包含了重写相关的方法，那么该接口的函数类型就是可以有效重写所有继承下来的抽象方法的单个方法的类型。在该例中，如果执行如下代码：

```
interface Foo extends Foo1, Foo2 {}
```

那么 Foo 的函数类型就是如下方法类型：

```
public void bar(List arg);
```

这些问题对于单个抽象方法这种简单情况来说是没有什么影响的，不过在遇到像 Comparator 这样的特殊情况时，能够理解背后的原因还是非常有价值的。

2.7.2 匹配函数类型

针对目标类型提供的函数式接口类型对 lambda 进行类型检查要求 lambda 表达式兼容于接口的函数类型。例如，如下赋值：

```
UnaryOperator<Integer> b = x ->x.intValue();
```

可以编译通过，因为lambda表达式兼容于UnaryOperator<Integer>的函数类型。下面列出了保证兼容性的条件[5]：

参数数量：lambda 与函数类型要有相同数量的参数。

参数类型：如果 lambda 表达式的类型是显式定义的，那么其类型就要与函数类型的参数相匹配；如果 lambda 的类型是隐式定义的，那么对于返回类型检查来说(下面会介绍)，其参数类型会被认为与函数类型的相同。

返回类型：

- 如果函数类型返回 void，那么 lambda 体必须是一个语句表达式(也就是说，表达式可以用作语句，就像方法调用或赋值一样)，例如：

```
(int i) -> i++;
```

---

[5] 本节实际上是 *The Java Language Specification*(http://docs.oracle.com/javase/specs/jls/se8/html/)Java 8 版中 15.27.3 节的简化版。

注意这里语句表达式的值被丢弃了；此外，还可以使用没有返回值语句的块体[6]，例如：

```
(Thread t) ->{ t.start(); }
```

- 如果函数类型具有非 void 的返回值，那么 lambda 体就必须返回一个与赋值兼容的值。例如，下面这个 lambda 体返回一个 int 值，它会被赋给 UnaryOperator 的函数类型的 Integer 结果。

```
UnaryOperator<Integer> b = x ->x.intValue();
```

对于方才给定的兼容性条件来说，我们还需要添加进一步的条件来确保 lambda 表达式能够编译通过：

抛出类型：lambda 表达式可以抛出检查的异常，前提是函数类型声明抛出了该异常或是其父类型异常。

最后一个条件可能会导致问题。例如，假设我们想要声明一个方法来集中处理 File 的无参数方法所实现的各种不同的 I/O 操作所造成的 IOException。首先要声明：

```
<U> U executeFileOp(File f, Function<File,U>fileOp) { ... }
```

不过 Function 的函数类型并没有声明任何异常，因此下面这种调用：

```
executeFileOp(f, File::delete);
```

无法编译通过。相反，我们需要一个自定义的函数式接口：

```
@FunctionalInterface
interface IOFunction<T,R> {
```

[6] 后面会经常使用“拥有值”或“拥有流”这两个术语来描述会返回值或流的方法。

```
    R apply(T t) throws IOException;
}
```

这样，我们就可以根据意愿声明 executeFileOp 了：

```
<U> U executeFileOp(File f, IOFunction<File,U>fileOp) {
    try {
        returnfileOp.apply(f);
    } catch (IOException e) {
        // centralized exception handling
    }
}
```

总结一下：lambda 可以像其他方法一样以两种相同的方式处理抛给它的异常——它可以隐式将其声明给调用者，也可以在 catch 块中处理。由于库函数式接口都没有声明任何异常，因此如果实现了用户定义的函数式接口，那么 lambda 只会将异常传递给调用者，就像示例中演示的那样。与之相反，我们将会在 5.3 节中看到，对于流的处理来说，我们需要在 lambda 中采用迂回的方式处理检查的异常，因为流处理操作的参数都是库函数式接口。简而言之：在 Java 8 lambda 中，检查的异常的处理是一个问题。

2.8 重载解析

对于 2.7.2 节的兼容性标准来说，当函数式接口类型可以明确下来时，lambda 表达式的目标类型就非常清楚了。2.5 节列出的大多数目标类型上下文都提供了明确的上下文。但遗憾的是，当方法拥有不同的重载变体时，lambda 表达式最常用的上下文(方法调用站)也是最容易出现问题的。本节介绍的难题并不常见；但当其出现时却会导致令人沮丧的编译问题。如果你在设计 API，而

用户依靠你来避免这个问题，那就得好好研究这个话题了——凡事预则立！

下面对重载解析所面临的挑战做一个小结：调用不同参数个数的重载方法通常很容易区分，不过在参数个数相同的两个重载方法之间选择时则要求编译器知道所提供的参数类型。对于无类型的 lambda 来说，这就是个问题，因为在没有目标类型的情况下是无法推断出其类型的，而目标类型是由方法声明提供的，因此只有在完成重载解析后才能知晓这一切！我们可以通过施加一些规则来打破这个循环，要么就是让类型推断过程违反直觉和不可预测，要么就是强制用户提供 lambda 参数类型。为了避免这些不合需要的行为，设计者最终决定让重载解析过程在不确定的情况下更加宽容一些并设计出新的 API，从而可以通过隐式类型的 lambda 高效地实现重载解析。

2.8.1　lambda 表达式的重载

在探究使用类型推断所会遇到的困难之前，我们先来看看在简单直接的情况下其工作方式是怎样的。第 1 章(1.4 节)介绍了方法 Comparator.comparing，给定一个键抽取器，它会根据其创建一个 Comparator。这是对 comparing 的一个重载(为了可读性，这里略掉了大多数类型边界)：

```
public static <T,U extends Comparable<U>>
      Comparator<T>comparing(Function<T,U>keyExtractor)
```
❶

假设我们想要根据长度对字符串进行排序。给定明确的目标类型，例如赋值语句，comparing 会接收一个无类型的 lambda 表达式，然后通过它创建一个比较器来实现目标：

```
Comparator<String> cs = Comparator.comparing(
      s ->s.length());                                    ❷
```

这个高度简化的程序解释了❷中的lambda类型是如何解析的。首先，编译器要选择一组comparing重载方法，并根据这些方法检查lambda表达式。由于另一个重载方法接收两个参数，因此这组重载方法只包含❶。既然已经选择好了重载方法，那么❶的返回类型与❷的目标类型的匹配就会让编译器推断出在该例中T就是String。现在，lambda表达式可以与Function<T, U>的函数类型相匹配了：

```
public U apply(T t)
```

既然传递给 lambda 的参数类型 T 已经确定为 String，那么 String.length 的返回类型 int(会被装箱为 Integer)就可以替代 U 了。现在，所有类型都是已知的，❶中绑定在 U 上的类型是 Integer(它确保 lambda 的结果实现了 Comparable)。一切都是一致的，调用也可以编译通过。

到目前为止一切都很好。但是假设 Comparator 声明了 comparing 的另一个重载：

```
// removed from the JDK
public static <T>
      Comparator<T> comparing(ToIntFunction<T>            ❸
      keyExtractor)
```

在 Java 8 开发期间，这种重载确实存在。如果再次对❷进行类型检查，我们就会看到将这种情况取消的原因所在。遵循之前相同的流程，编译器首先会尝试选择针对于类型检查的重载方法。规则是这个行为必须在无类型 lambda 的类型检查之前进行(某些无类型的 lambda 可以在方法重载解析前进行类型检查，不过这种

行为被放弃了，因为设计者发现这么做会导致整个流程变得更加不一致和混乱)。

假定使用这个规则，那么提供给方法重载解析的信息就会少之又少。编译器无法使用赋值上下文所提供的目标类型，因为另一个重要规则要求无论上下文是什么，带有给定参数的方法调用必须被解析为相同的重载方法。它可以对方法调用使用其他参数，不过在该例中是没有参数的。它可以使用来自于 lambda 表达式的信息，不过对于兼容性条件来说，像这样的无类型 lambda 只能提供所接受的参数个数以及是否返回值等信息。在该例中，这些信息起不到任何帮助作用：lambda 接受一个参数并返回一个值，并且与两个候选的函数式接口的函数类型相匹配。这样，整个流程就卡住了，编译器会报错，给出的消息是“reference to comparing is ambiguous”。

那么我们该如何避免这种情况发生呢？如果 lambda 的参数类型是已知的，那么编译器就可以根据该信息，使用本节一开始介绍的算法来检查两个候选的函数类型。由于参数类型是 String，因此编译器会发现它们都与 lambda 兼容。接下来，编译器根据哪一个会返回最具体的结果来从这两个候选者当中进行选择：ToIntFunction 返回一个 int，Function<String, Integer>返回一个 Integer，前者与 String.length 的返回类型更为接近。因此，下面的调用：

```
    Comparator<String> cs = Comparator.comparing((String s)
->s.length());
```

可以编译通过，并被解析为❸。对于 Comparator.comparing 来说，Java 8的设计者们不希望强迫用户像这样提供显式的 lambda 类型，因此它们使用新方法 comparingInt、comparingLong

与 comparingDouble 来替换接受的 ToIntFunction、ToLongFunction 与 ToDoubleFunction 的重载方法。对于其他 API 来说，如果函数式接口的参数个数不同，或是向重载方法提供的参数无二义性，那就可以提供不同的重载方法。如果你在设计 API，那么用户就会感谢你选择了这些实现方式。

2.8.2 方法引用的重载

方法与构造器引用又引入了新的难题：如果它们指向的方法或构造器是泛型的或重载的，那就是不精确的，因为其语法与显式的 lambda 类型是不一样的，因此你不能指定想要使用的精确类型。注意这是一种相对来说不太常见的情况：实际上，大多数方法既不是重载的，也不是泛型的。本节最后将会看到，可以通过使用类型化的 lambda 避免不精确的方法引用所带来的问题；此外，这也不是该问题的唯一解决方案。

举一个不精确的构造器引用的示例，Exception::new 可以指向如下两个 Exception 构造器：

```
public Exception()
public Exception(String message)
```

第 1 个匹配 Supplier<Exception>的函数类型，第 2 个匹配 Function<String, Exception>的函数类型。如果声明如下方法重载：

```
<T> void foo(Supplier<T> factory)
<T,U> void foo(Function<T,U> transformer)
```

那么调用：

```
foo(Exception::new);
```

将会出现编译错误，错误消息是“reference to foo is

ambiguous”。不过，虽然我们无法像显式指定 lambda 类型那样让构造器引用更加精确，但还是有解决办法的：由于 foo 是泛型的，因此我们可以提供类型见证者来指定想要实例化的类型。该语法要求显式指定接收者：

```
this.<Exception>foo(Exception::new);
this.<String,Exception>foo(Exception::new);
```

当然，这么做只能解决具有不同数量的类型参数的泛型方法。假设我们想要调用如下两个重载方法之一：

```
void bar(IntFunction<String> f)
void bar(DoubleFunction<String> f)
```

在调用时传递指向String.valueOf的引用，这是一个静态方法，其重载版本分别匹配IntFunction<String>和DoubleFunction<String>。这样，只需要进行强制类型转换即可解决问题：

```
bar((IntFunction<String>) String::valueOf);
bar((DoubleFunction<String>) String::valueOf);
```

String::valueOf 的重载结果是其不同的函数类型匹配单个参数的可能类型，而后者用于区分 bar 的不同重载版本，因此问题就出现了。虽然这种情况可能会出现，但并不常见。如果真的出现了，那么请记住你总是可以使用相应的 lambda 表达式：

```
bar((double i) ->String.valueOf(i));
bar((int i) ->String.valueOf(i));
```

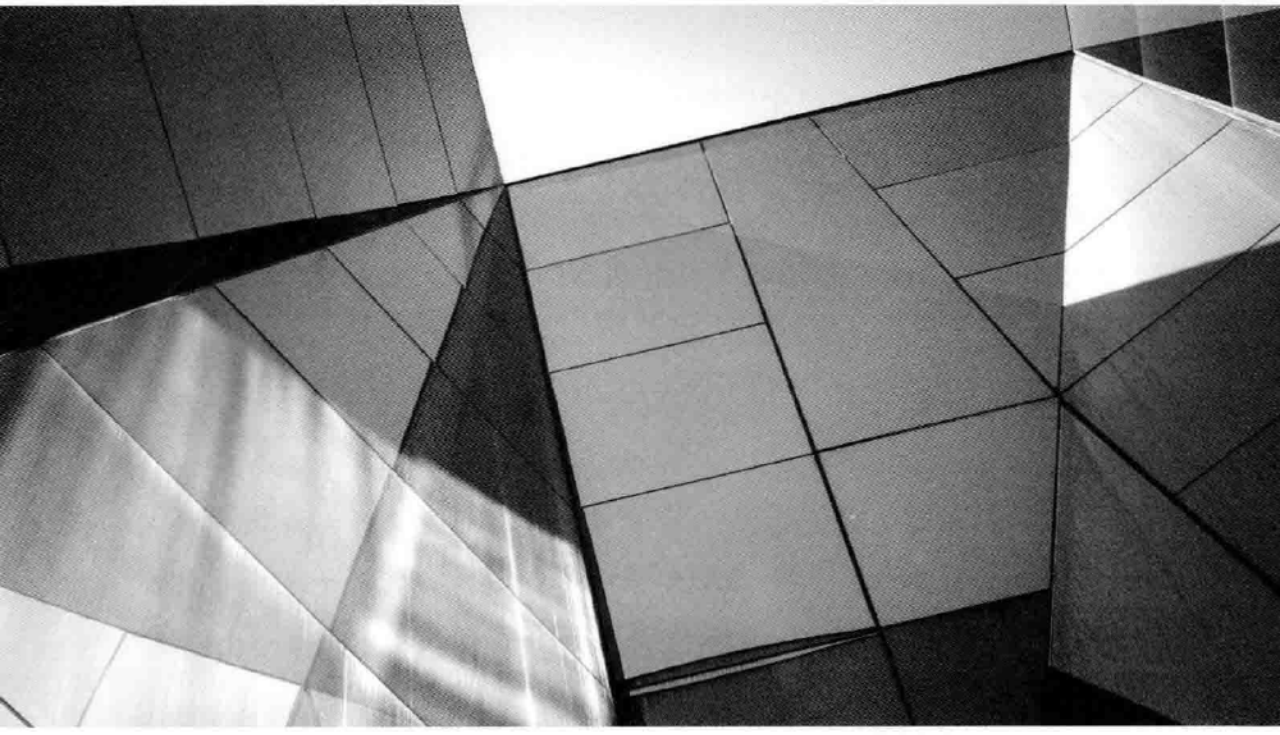
总结一下：如果类型推断或是重载解析失败了，那么你就需要提供更多的类型信息。有多种方式可以做到这一点，侵入性最小的方式是找到相应的类型化 lambda 表达式或是精确的方法引用。如果不行，那就提供一个类型见证者。从代码风格的角度来

看，强制类型转换应该是不得已而为之的手段，能不用尽量不用。

2.9 小结

到目前为止，我们已经介绍了为引入 lambda 需要做的所有语法变更；Java 8 的另一个主要变化是引入了默认方法，通过允许接口定义行为来支持 API 的演化。第 7 章将会介绍默认方法。本章介绍的一些细节看起来可能很复杂，不过总的来说，背后的想法是非常简单的，至少要比最终版本形成之前的中间阶段简单不少。

语言的演进总是困难的，因为每个新特性都会与现有特性存在或多或少的交互；复杂性就是个典型，这在本章中已经介绍过了，装箱/拆箱与无类型的聚合表达式对方法重载解析引入了很多复杂性。另外就是编写可以抛出检查的异常的 lambda 表达式是很困难的。在 lambda 语法的设计与实现中曾考虑过很多折中方案，其中与类型推断相关的方案最为引人关注，主要是因为它有时不如你想象的那么强大。这是因为在设计类型推断规则时最重要的推动力就是保持其简单性和一致性，这个想法也指导着整个语言的设计。



第3章

流与管道介绍

第 1 章介绍了将 lambda 引入 Java 中的两个主要动机。这两个动机(更好的代码，更轻松的并行性)在介绍流时走到了一起以进行集合的处理。虽然 lambda 表达式对于流编程是必不可少的，但这还不够：要想高效使用流，我们还需要一些特定于流编程模型的新想法。本章将会介绍流底层的基本理念，通过关于流操作的直观示例帮助大家理解流的工作方式。后续章节将会进一步探索更加理论化的主题，介绍流代码是如何简化常见的集合处理任

务的，最后研究一下不同场景下串行流与并行流的性能。

我们将要介绍的每个流操作本身看起来可能不是非常强大。Stream API 的强大之处在于可以将这些操作组合起来实现各种各样的功能。据此，其设计从计算机的历史长河中吸取了很多经验。与其最为接近的前辈就是所谓的“Unix 哲学”，*The Unix Programming Environment*(Kernighan 与 Pike 合著，1984)的作者写道：

> 其核心理念就是系统的功能应该更多地源自于程序之间的关系而非程序本身。很多 UNIX 程序本身都只会完成一些小功能，但与其他程序组合起来就会变成非常棒的工具。

如果熟悉 Unix 管道中该哲学的实现，那么你就会发现这清晰地反映在流的中间操作上。不过，更广泛地说，组合原则在整个 Java 8设计的改变上处于中心位置：在1.4节中，我们看到了 lambda 表达式是如何促成了更加细粒度的设计、组合型更强的操作，稍后将会看到它对流收集器设计的影响。

在后面探究流的技术细节时，我们不会遗漏掉其重要构想：对常见聚合操作的表现力更强的表述。流设计是否成功要依靠它距离对常见业务逻辑更清晰的表达这一目标有多远来衡量。

3.1 流基础

1.2 节中曾介绍过流，当时将其作为可选的有序值序列。从操作角度来看，流与集合是不同的，因为流不存储值；流的目的是处理值。例如，考虑一个将集合作为源的流：创建流并不会导致数据流动；当终止操作需要值时，流会从集合中获取值将其提供；

最后，当所有集合值都被流提供后，流就被耗尽了，并且无法再次使用。不过这与空是不同的：流从来不会持有值。对于源不是集合的流来说，其行为也是非常类似的：例如，我们可以通过如下代码生成并打印出 2 的前 10 次幂：

```
IntStream.iterate(1, i -> i*2)
    .limit(10)
    .forEachOrdered(System.out::println);
```

不过后面将会看到，方法 iterate 会生成一个无限流，lambda 表达式所表示的函数的调用次数取决于后续处理所需的值(在该示例中是 9 次)。

流背后的中心思想是延迟计算：直到需要时才计算值。Java 程序员对延迟计算已经有所了解[1]：每天都会用到的迭代器，无论是显式使用还是隐式使用，都具有这个特性。迭代器的创建并不会导致任何值被处理；只有调用其 next 方法时才会使其从集合中实际返回值。从概念上来说，流与迭代器非常类似，不过流具有一些重要的改进：

- 流以一种对客户端更加友好的方式处理无元素的情况。迭代器只会通过从 hasNext 调用中返回 false 的方式来发出没有元素的信号，这样客户端在每次请求元素时就必须进行测试。这种交互天生就容易出错，因为调用 hasNext 与 next 之间的时间间隔可能会出现线程干扰问题。此外，迭代器强制以序列化的方式处理元素，其实现造成了客户端与库之间复杂且低效的交互。

[1] 当然了，所有程序员都应该知道延迟计算，因为它是程序员所具有的 3 个伟大品质的第 1 个：懒惰(在英文中，延迟与懒惰都是 lazy，作者这里使用了双关语)、急躁和傲慢((Larry Wall 等著，*Programming Perl*，O'Reilly,2012)。

- 迭代器总是以确定的顺序来处理值，与之不同，流可以是无序的。第 6 章将会深入介绍这一点；现在，你只需要知道，如果我们不关心值的顺序，那么就可以通过并行流进行优化。
- 流具有方法(中间操作)，这些方法会接受行为性参数(对流的转换)，然后返回转换后的流。这样就可以将流链接到管道中，就像在第 1 章看到的那样，这不仅可以提供流式编程，还能在极大程度上提升性能。稍后将会详细介绍中间操作。
- 流会保留关于源的属性的信息，例如源值是否是有序的，总数是否是已知的，这样就可以对值处理进行优化，而迭代器则是做不到这一点的，除了值本身外，迭代器并不保留任何其他信息。

延迟计算的一个重要优势可以通过 Stream 的“搜索”方法一探究竟：findFirst、findAny、anyMatch、allMatch 与 noneMatch。它们都称为“短路”操作符，因为使用了这些方法就没必要处理流中的所有元素了。例如，anyMatch 需要找到一个满足其谓词(boolean 值函数)，让流处理完成的单个流元素，allMatch 则需要找到一个不满足其谓词的流元素。这无须生成和处理不必要的元素，能够极大地降低工作量，对于无限流来说，我们只需要将延迟计算与短路操作符组合起来就可以完成流的处理。当然了，从原理上来说，延迟计算的这种优势也可以通过迭代器(对于集合处理来说)或是显式循环(对于生成器函数来说)实现出来，不过使用 Stream API 编码更容易读，也更容易写。

延迟计算还提供了另外一个好处：可以将多个逻辑运算合并到一起并统一应用到数据上。回忆一下第 1 章的第 1 个用于演示管道概念的代码：

```
OptionalDouble maxDistance =
    intList.parallelStream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p ->p.distance(0, 0))
        .max();
```

这种流式风格自然又易读，不过要想理解它则需要考虑到隐式的延迟计算。出于演示的目的，如果将管道拆分为流声明与终止操作调用，那就更容易看清楚这一点了：

```
DoubleStream ds =
    intList.parallelStream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p ->p.distance(0, 0));

// the pipeline has now been set up, but no data has been processed
yet

OptionalDouble maxDistance = ds.max();
```

将终止操作调用分离开有助于大家的理解：它会直接调用行为型参数的转换代码，而这些代码是一次性执行的。此外，由于对于每个元素来说，这些操作都是由单个线程执行的，因此依赖于代码与数据局部性的优化是可以做到的。

该模型与大多数集合处理都存在着明显差别，后者通常情况下会由一系列通路实现，每个通路都会转换集合中的每个元素，然后将结果存储到新的集合中。我们在第 1 章看到如何将一个循环序列转换为相应的流操作序列。流代码既汇聚又高效，因为它是通过将单独的循环操作融合到一起而实现的。

3.1.1　面向并行的代码

延迟值序列在编程中并不是什么新概念；其在 Java 中的实现

的特别之处在于它将这个概念扩展了，从而包含了对元素的并行处理。虽然顺序处理依然是个非常重要的计算模型，不过它不再是唯一一个参考模型了：第 1 章曾经介绍过，并行处理已经变得非常重要了，如果代码的运行结果与如何执行没有关系，无论是串行还是并行，那么我们就需要重新思考计算模型了。如果采用这种方式，那么当未来需要转向并行执行时，我们的代码(甚至更为重要的，我们的代码风格)就无须变更。Stream API 鼓励大家通过这种视角看待问题；理解它的关键在于要知道所有操作都有相应的串行模式与并行模式，只要程序员遵循一些简单的规则即可。事实上，这种思考处理模式的方式可以从下面这个事实中推断出来，即单个 Stream 接口既表示串行流，也表示并行流；API 设计者们曾经考虑过在单独的接口中分别提供串行方法与并行方法，不过后来被否定了。

注意，这并不意味着操作在每种模式下总是会产生出完全相同的结果：例如，forEach 就可以在不确定的顺序下执行。在串行模式下它可以保存顺序，但这么实现是有问题的，如果依赖于此就会出错；forEach 与其他非确定性操作在串行与并行模式下的顺序都是不确定的。与之相反，在这两种模式下顺序确定的操作则会保证以确定的顺序执行。之所以会提供 forEach 之类的非确定性操作，原因在于不必要的确定性是有着很高的性能代价的[2]。如果选择相应的确定性版本(例如 forEachOrdered)，那就要清楚地认识到顺序是问题之所需，而非仅仅是随意的选择。

这也有助于解释该如何选择定义在 API 中的操作，可以避免

---

[2] ReentrantLock 类就是一个众所周知的示例，可以通过公平(确定的)或非公平(不确定)的线程调度策略来创建它。公平锁的效率至少要比非公平锁低一个数量级，在实际情况下很少使用，同时公平调度策略使用得也很少(Brian Goetz 等著，*Java Concurrency in Practice*，Addison-Wesley，2006。)

在串行与并行模式下结果不同的情况出现。一些“显而易见”的操作隐藏了很深的串行偏见，例如“takeWhile”操作，在其他平台上，它会处理一个流，直到遇到了某个标记为止。虽然要想并行化这个操作也不是不可能，不过实现起来代价非常大，无论是设计还是执行上，因此优先级被降低了。

思维上的转变还可以再来组织一下：迭代的程序包含了两类信息：要做什么，以及如何做。将这两个概念解耦就得到了一种开发模型，我们编写面向并行的代码，只需要指定函数式行为即可，然后再单独指定其执行模式，在理想的情况下，会将执行的实现委托给库来做。这是个巨大的变化：对于我们大多数人来说，串行模式已经在脑海中根深蒂固了，要想摆脱它的影响还需要付出一些努力才行。不过，这种努力是值得的，最终的程序不仅不会过时，我们的代码还会变得更加明晰、更加简洁、可维护性更好。

3.1.2　原生流

Java 5 中引入的自动装箱与自动拆箱特性使得程序员无须考虑原生类型值与相应的包装类型值之间的差别。编译器经常会发现在方法调用或是赋值时，本来需要的是包装值，但实际提供的却是原生值(反之亦然)，它会自动进行转换。这是非常便捷的，并且有助于编写可读性更好的程序；泛型集合类现在可以包含原生值了，对于 Java 基本的面向对象编程模型来说，包装类型要比原生类型更加适合。不过，这会导致频繁的装箱与拆箱操作，对性能造成很大的影响。例如，在如下看似没有问题的代码中增加变量 i 的值：

```
Optional<Integer> max = Arrays.asList(1,2,3,4,5).stream()
    .map(i -> i + 1)
    .max(Integer::compareTo);
```

这会在每次增加 i 值之前分别执行 Integer 的方法 intValue 与 valueOf。我们希望在处理大规模集合值的应用中避免这种开销：一种方式就是定义流，其元素是原生值而非引用值。除了性能上的改进外，这么做还会带来其他好处，对于只处理数字的方法，如 sum，以及创建包含一系列数字的流来说都很有用。例如，借助于类型 IntStream(表示原生 int 值的流)，上述代码可以写成下面这样：

```
OptionalInt max = IntStream.rangeClosed(1,5)
   .map(i -> i + 1)
   .max();
```

这么做会改进可读性(使用了专门的 range 与 max 方法)与效率(不再需要装箱与拆箱操作)。6.6 节针对各种范围大小给出了这两个代码片段之间的性能差别。对于足够大的数据集来说，未装箱代码的速度要比相应装箱的代码快一个数量级。

原生流类型有 IntStream、LongStream 与 DoubleStream，这取自最常用的数字类型，成本收益也是最好的。如果应用需要其他数字类型的流，那么下面这些类型也是支持的：float 值可以放到 DoubleStream 中；char、short 与 byte 值可以放到 IntStream 中。原生流的 APIs 都很相似，并且所有原生流都与本章介绍的引用流类型 Stream 很相似。首先，我们需要知道可能发生的流类型转换：

- 原生流类型 IntStream 与 LongStream 拥有方法 asDoubleStream，IntStream还有方法asLongStream，这会导致每个原生值的范围变大，例如：

```
DoubleStream ds = IntStream.rangeClosed(
   1, 10).asDoubleStream();
```

- 对于装箱来说，每个原生流类型都有一个方法 boxed，它会返回恰当的包装类型的一个 Stream，例如：

```
Stream<Integer> is = IntStream.rangeClosed(1, 10).boxed();
```

- 对于拆箱来说，会通过调用恰当的映射转换操作(本节后面将会对其进行详细的介绍)，并提供恰当的拆箱方法将包装值的 Stream 转换为原生流。例如，如下代码创建了一个 Stream<Integer>，然后将其转换为 IntStream：

```
Stream<Integer> integerStream = Stream.of(1, 2);
IntStream intStream =
integerStream.mapToInt(Integer::intValue);
```

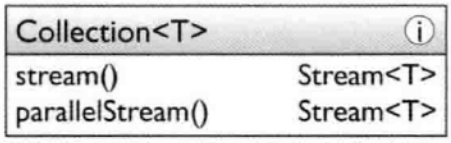
3.2　剖析管道

流的真正威力是通过创建并组合起来的管道得以实现的。我们已经介绍了管道的各个阶段：它源自流的源，随后通过中间操作进行一系列的转换，最后在终止操作中停止。在本章的后续内容中，我们将会详细介绍管道的每一阶段，让大家对流编程的可能性有一个清晰的认识；后续章节将会介绍赋予流强大威力的各个特性。

3.2.1　开始管道

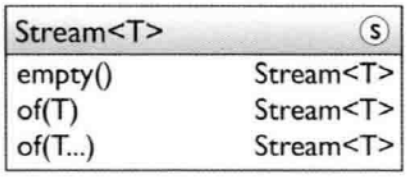
到目前为止，我们给出的流处理示例中的数据都来自于集合。事实上，流处理模型可以应用于各种数据块，基于这一优势，平台库中的很多能够生成数据块的类现在都可以创建提供数据的流。不过，本章重点关注于流的工作方式，因此第 5 章才会详细介绍平台库中完整的流创建方法的列表。目前，我们只需要使用 Collection 的流方法以及流接口中的工厂方法：

- java.util.Collection<T>：该接口中的默认方法也许是最常使用的生成流的方式[3]：



parallelStream的契约表示其返回的是一个“可能的并行Stream”。我们将会看到，以并行形式展示其数据是集合的职责，并非每个集合都会提供这个功能。虽然这会影响到性能，但对函数式行为却不会造成任何影响。第 6 章将会详细介绍这一点。

- java.util.stream.Stream<T>：该接口公开了大量的静态工厂方法，并且带有默认实现。本章将会使用 Stream.empty 以及 Stream.of 的两个重载方法(原生流类型也有类似的方法)：



这些方法足以让我们能够开始探索流与管道的特性了；第 5 章将会探索平台库中的其他流方法。

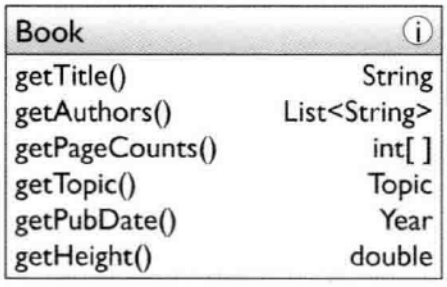
3.2.2 转换管道

流创建后，管道的下一阶段包含了大量的中间操作(也可能什么操作都没有)。如前所述，中间操作是延迟进行的：它们只会在

[3] 前面已经给出了 API 图的约定。简而言之：右上角的图标表示类图中包含的是静态方法还是实例方法；泛型类型的通配符绑定被略掉了；方法签名中未声明的泛型类型可以认为是方法类型参数；类图中只显示了类与接口中所选的方法。

终止管道操作出现时才计算其所需要的值。

本章与接下来的两章将会探索 Stream API，并且给出一些使用示例。我刚搬到一所公寓，公寓的书架上有很大的空间，让我能够将过去几十年来购买的好几百本书好好整理一下。我所规划的系统会将这个图书馆建模为 Book 对象的一个 Collection：



为了理解示例，我需要解释一下其中的属性：

- Topic是个枚举，其成员有HISTORY、PROGRAMMING等。
- 属性 pageCounts 指的是由多卷标题构成的卷的页数。其类型是数组，不过大多数情况下我们不推荐使用数组，Java 数组是一种遗留类型，在可能的情况下最好通过 List 实现将其替换掉，不过由于在维护遗留代码时还会经常遇到原生类型的数组，因此我们需要熟悉流是如何处理数组的。
- 一本书的多个版本，它们具有相同的标题和作者，如果出版日期不同，则可以在图书馆中并存。
- 书的高度对我来说非常重要，我的书架有不同的高度，因此在决定将不同主题的书放置在何处时要考虑到书的高度。

在探索不同的流操作时，使用一些具体的示例会很有帮助。下面是我的图书馆中 3 本书的声明：

```
Book nails = new Book("Fundamentals of Chinese Fingernail Image",
    Arrays.asList("Li", "Fu", "Li"),
```

```
        new int[]{256},          // pageCount per volume
        Year.of(2014),           // publication date
        25.2,                    // height in cms
        MEDICINE);

    Book dragon = new Book("Compilers: Principles, Techniques and
Tools",
        Arrays.asList("Aho", "Lam", "Sethi", "Ullman"),
        newint[]{1009},
        Year.of(2006),           // publication date (2nd edition)
        23.6,
        COMPUTING);

    Book voss = new Book("Voss",
        Arrays.asList("Patrick White"),
        newint[]{478},
        Year.of(1957),
        19.8,
        FICTION);

    Book lotr = new Book("Lord of the Rings",
        Arrays.asList("Tolkien"),
        newint[]{531, 416, 624},
        Year.of(1955),
        23.0,
        FICTION);
```

在后续的示例中，持久化机制被占位符变量 library 所代替，其声明为 List<Book>。图 3-1 展示了使用流来处理图书馆的代码示例；本章后面会对这些代码进行说明，这里将其放到了一起的目的在于让你对使用流处理有个整体认识。

本节将会介绍各种中间操作的动作，我们会通过示例了解每个动作是如何用在操作中以处理图书馆的。虽然主要介绍的是对引用流类型 Stream 的操作，但其中的想法也可以应用到原生流类型上；对于中间操作来说，其 API 与 Stream 非常接近。

流处理的示例

只包含计算过的图书的流：

```
Stream<Book> computingBooks = library.stream()
    .filter(b ->b.getTopic() == COMPUTING);
```

图书标题的流：

```
Stream <String> bookTitles = library.stream()
    .map(Book::getTitle);
```

Book 的流，根据标题排序：

```
Stream<Book> booksSortedByTitle = library.stream()
    .sorted(Comparator.comparing(Book::getTitle));
```

使用这个排序流创建一个作者流，根据图书标题排序，并且去除重复的：

```
Stream<String> authorsInBookTitleOrder = library.stream()
    .sorted(Comparator.comparing(Book::getTitle))
    .flatMap(book ->book.getAuthors().stream())
    .distinct();
```

以标题的字母顺序生成前 100 个图书的流：

```
Stream<Book> readingList = library.stream()
    .sorted(Comparator.comparing(Book::getTitle))
    .limit(100);
```

除去前 100 个图书的流：

```
Stream<Book> remainderList = library.stream()
    .sorted(Comparator.comparing(Book::getTitle))
    .skip(100);
```

图书馆中最早出版的图书：

```
Optional<Book> oldest = library.stream()
    .min(Comparator.comparing(Book::getPubDate));
```

图书馆中图书的标题集合：

```
Set<String> titles = library.stream()
    .map(Book::getTitle)
    .collect(Collectors.toSet());
```

图 3-1　本章 Stream 操作示例

过滤

方法 filter 可以有选择地处理流元素：

Stream<T>	ⓘ
filter(Predicate<T>)	Stream<T>

其输出是输入流中满足提供的 Predicate 的那些元素。例如，我们可以通过构建下面这样的流来找出计算机图书：

```
Stream<Book> computingBooks = library.stream()
.filter(b ->b.getTopic() == COMPUTING);
```

图 3-2 演示了 filter 的动作。在(a)部分中，两个 Book 元素到达了输入流，先是 nails，然后是 dragon；在(b)中，输出流只包含了 dragon，因为只有它的主题是 COMPUTING。

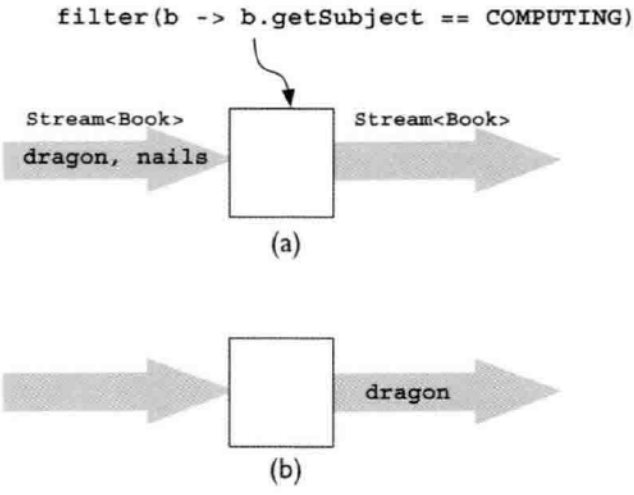


图 3-2 Stream.filter 的工作原理

映射

方法 map 会通过提供的 Function<T, R>转换每个流元素。

Stream<T>	ⓘ
map(Function<T,R>)	Stream<R>

其输出是一个流，包含了对输入流中每个元素应用了 Function 后的结果。例如，可以通过它创建出版日期的 Stream:

```
Stream<Year> bookTitles = library.stream()
.map(Book::getPubDate);
```

图 3-3 演示了 map 的动作。在(a)中，输入元素是与上面相同的两个 Book 实例；在(b)中，结果流中包含了对 Year 对象的引用，这是通过对每个实例调用 getPubDate 得到的。

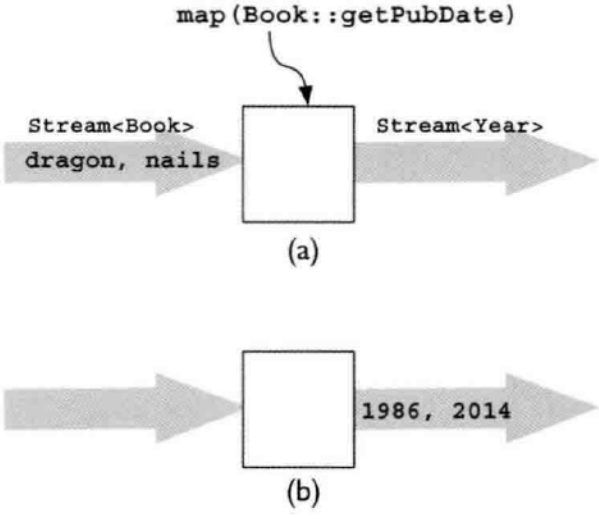


图 3-3　Stream.map 的工作原理

方法mapToInt、mapToLong与mapToDouble对应于map。它们会通过ToIntFunction<T>、ToLongFunction<T>与ToDoubleFunction<T>的实例将引用类型流转换为原生流，每个转换都会接收一个T并返回一个原生值。

Stream<T>	ⓘ
mapToInt(ToIntFunction<T>)	IntStream
mapToLong(ToLongFunction<T>)	LongStream
mapToDouble(ToDoubleFunction<T>)	DoubleStream

我们之前曾看到过这些方法可用于拆箱一系列包装值。再举一个例子，我可以计算出图书馆中所有图书的作者总数，如下代码所示：

```
int totalAuthorships = library.stream()
    .mapToInt(b ->b.getAuthors().size())
    .sum();
```

按照这种方式转换为原生流可以充分利用其更好的性能以及专门的数学终止操作，例如 sum。

Stream API 支持这 4 种 Stream 类型之间的相互转换。这样，除了这里介绍的 Stream 方法外，每个原生流都有 3 个转换 map 操作，用于转换为其他 3 种类型。例如，除了 map 之外，IntStream 还有转换操作 mapToLong、mapToDouble 以及 mapToObj。

一对多映射

实现上一个示例的另外一种方式(不过性能要低一些)就是将 Book 的流转换为 Author 的流，每个都表示一个作者关系。接下来只需要使用终止操作 count 来找出流中元素的数量。不过 map 并不适合于这个目的，因为它会对输入流的元素执行一对一的转换，而该问题需要将单个 Book 转换为输出流中的几个 Author 元素。我们需要的操作会将每个 Book 映射为一个 Author 的流，即 book.getAuthors().stream()，然后将生成的一系列流压平为针对所有图书的单个 Author 的流。

这就是操作 flatMap：

Stream<T>	ⓘ
flatMap(Function<T, Stream<R>>)	Stream<R>

对于该示例来说，代码如下所示：

```
Stream<String> authorStream = library.stream()
    .flatMap(b ->b.getAuthors().stream());
```

图 3-4 展示了其工作原理。在(a)中，相同的两个 Book 实例位于输入流中，在(b)中，每个 Book 会被映射为一个 String 流，在(c)中，这些独立的流会被注入输出流中。

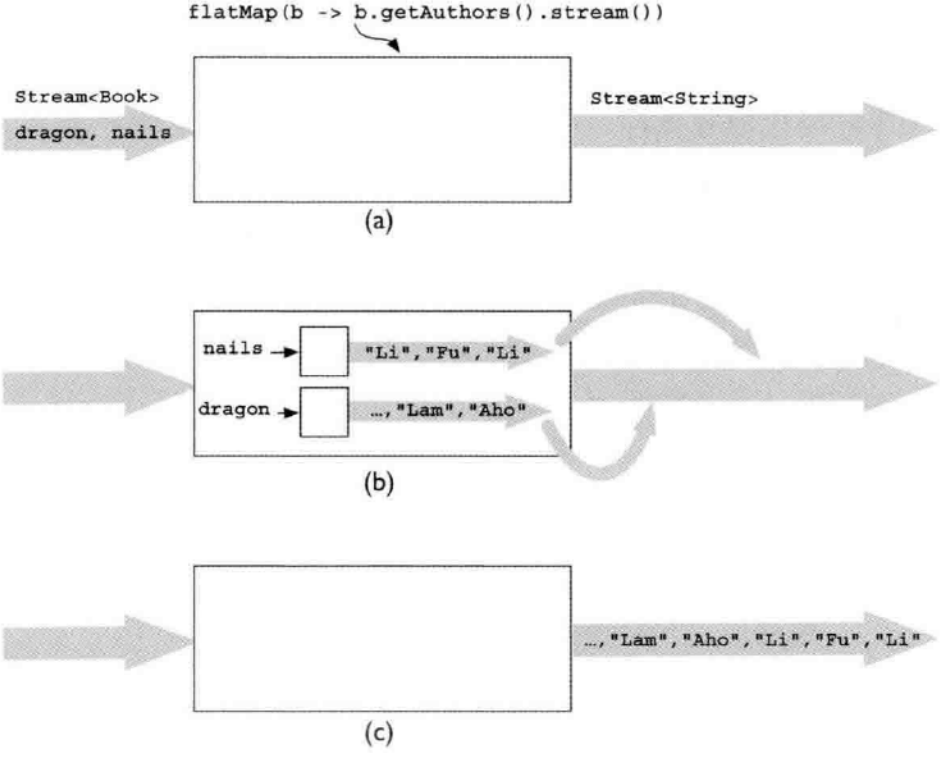


图 3-4　Stream.flatMap 的工作原理

类似于与 map 对应的用于转换为原生流的方法，也存在着原生转换方法：flatMapToInt、flatMapToLong 与 flatMapToDouble。例如，我们可以获得所有图书所有卷的总页数，方式是通过 IntStream.of 为每个 Book 创建一个单独的 IntStream，然后通过 flatMapToInt 将它们连接起来：

```
int totalPageCount = library.stream()
    .flatMapToInt(b ->IntStream.of(b.getPageCounts()))
    .sum();
```

原生流类型只有 flatMap；它们并没有用于类型转换的压平操作。

调试

正如我们在 3.1 节中看到的那样，调用管道的终止操作会导致其中间操作得到执行。这样，通常的单步调试技术就不适用于流了。Stream API 提供的另外一个操作 peek 与其他中间操作不同，其输出流会包含相同的元素，并且与输入流的顺序相同。peek 的目的旨在对处于管道中间位置的流元素执行处理；例如，我们要打印出每本书的标题，并且在将其发送给下游做进一步的处理之前向其传递一个过滤器(在该示例中就是将其放到一个 List 中，参见 3.2.4 节)：

```
List<Book> multipleAuthoredHistories = library.stream()
     .filter(b ->b.getTopic() == Book.Topic.HISTORY)
     .peek(b ->System.out.println(b.getTitle()))
     .filter(b ->b.getAuthors().size() > 1)
     .collect(toList());
```

如果熟悉 Unix 管道，那么你就会发现这非常类似于 tee，不过 peek 的通用性更强一些，它可以接收任何类型适当的 Consumer

作为参数。该方法用于对调试提供支持，由于其有副作用，因此不应该用于其他任何目的。3.2.3 节将会介绍其副作用与干扰性问题。

排序与去重复

操作 sorted 的行为与我们期望的一致：输出流包含输入流中的元素，并且是有序的。

Stream<T>	ⓘ
sorted()	Stream<T>
sorted(Comparator<T>)	Stream<T>
distinct()	Stream<T>

排序算法对于流来说是稳定的(参见第 6 章)；这意味着如果被排序的流是有序的，那么在输入与输出之间，具有相同键的元素之间的相对顺序会保持不变。

第 1 个 sorted 重载方法会使用自然顺序对对象进行排序。例如，我们可以使用它创建一个 Book 标题的流，并且按照字母顺序排序：

```
Stream<String> sortedTitles = library.stream()
    .map(Book::getTitle)
    .sorted();
```

第 2 个重载的sorted会接受一个Comparator；例如，静态方法Comparator.comparing会根据一个键抽取器创建一个Comparator：

```
Stream<Book> booksSortedByTitle = library.stream()
     .sorted(Comparator.comparing(Book::getTitle));
```

通过从一个键创建一个 Comparator，它提供了除了对流元素

按照自然顺序进行排序的另外的排序方式。注意，它没有限制你必须要按照自然顺序排序：另一个重载的 Comparator.comparing 接受一个针对抽取键的 Comparator，这样就可以对其使用不同的排序规则了。例如，为了根据作者数量对图书进行排序，可以写成下面这样：

```
Stream<Book> booksSortedByAuthorCount = library.stream()
    .sorted(Comparator.comparing(Book::getAuthors,
      Comparator.comparing(List::size)));
```

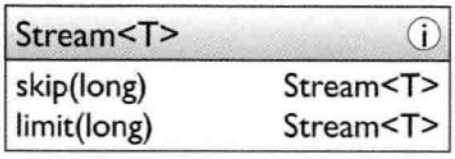
组中的第 2 个操作 distinct 会从流中删除掉重复的元素。其输出流中只包含输入元素中单次出现的元素——根据 equals 方法，所有重复元素都会被丢弃掉。例如，要想根据方才创建的排好序的 books 流来生成一个作者列表，并去除掉重复元素，我们可以写成下面这样

```
Stream<String> authorsInBookTitleOrder = library.stream()
    .sorted(Comparator.comparing(Book::getTitle))
    .flatMap(book ->book.getAuthors().stream())
    .distinct();
```

由于 equals 方法可能不会考虑到流元素的所有字段，因此“重复”可能不同于其他方法的结果。因此，与 sorted 相关的稳定性概念也适用于 distinct：如果输入流有序，那么元素之间的相对顺序就会保持下来。对于大量相同的元素来说，它会选择第 1 个；如果输入流无序，那可能就会选择任意一个元素(在并行管道中这是一个代价很低的操作)。

截断

截断有两个操作，它会对来自于流的输出采取一些限制措施：



操作 skip 与 limit 是双重的：skip 会丢弃掉前 n 个流元素，返回剩余元素，而 limit 则会保留前 n 个元素，返回的流中只包含这些元素。例如，我们可以根据标题的字母表顺序得到前 100 本图书，就像下面这样：

```
Stream<Book> readingList = library.stream()
    .sorted(Comparator.comparing(Book::getTitle))
    .limit(100);
```

当然，图书馆中的图书可能不到 100 本；在这种情况下，limit 返回的流中就会包含所有图书。与之类似，skip 可用于创建一个包含除了前 100 个元素的剩余元素的流：

```
Stream<Book> remainderList = library.stream()
    .sorted(Comparator.comparing(Book::getTitle))
    .skip(100);
```

3.2.3　非侵入性

在 Stream API 提供给程序员的众多特性中包含了并行操作的执行，甚至对于非线程安全的数据结构亦如此。这是个很有价值的特性，因此其使用也是有成本的。根据本节介绍的规则，其成本与编程模型有关。这并非随意的限制；事实上，只有在你以串行模式思考时才会感受到其限制。如果已经设计好了面向并行的代码，那么它们看起来就是非常自然的了，因为在你的思维方式中已经不再假设哪个线程会执行流操作的行为型参数，以及元素的处理顺序。你已经知道每个行为型参数都有可能在不同的

线程中执行，唯一的排序限制是由流的执行顺序所施加的(参见6.4 节)。

规则可以防止程序遭受到并行执行所要求的多线程的干扰。有时，框架本身会提供这种保证，就像收集器那样。不过对于流操作的行为型参数却没有提供这种保证。下面这个示例展示了如何不创建一个分类 map，并将每个主题映射到与该主题对应的图书列表中：

```
// behavioral parameter with state - don't do this!
Map<Topic,List<Book>> booksByTopic = new HashMap<>();
library.parallelStream()
  .peek(b -> {
    Topic topic = b.getTopic();
    List<Book> currentBooksForTopic = booksByTopic.get(topic);
    if (currentBooksForTopic == null) {
      currentBooksForTopic = new ArrayList<>();
    }
    currentBooksForTopic.add(b);
  booksByTopic.put(topic, currentBooksForTopic); // don't do
this!
})
.anyMatch(b -> false); // throw the stream elements away
```

构成peek的行为型参数的lambda表达式不是线程安全的。当并行执行时，错误就会出现：map entries会丢失(在我自己的Core 2 Duo机器上并行执行时，约有 1%会丢失)；调用currentBooksForTopic.add很有可能会抛出ArrayIndexOutOfBoundsException异常。可以将lambda体同步起来解决这个问题，但这么做是错误的，因为太笨拙且低效。正确的做法是要注意到行为型参数应该是无状态的。Stream API旨在提供安全且性能良好的并行解决方案，在这种情况下可以使用groupingBy收集器，下一节将会对其进行介绍。

如果从来不在自己的代码中使用并行流，那就无须考虑这些

问题，就像上一代程序员们错误地认为如果不使用线程，那么就无须考虑同步问题一样。随着库 APIs 在未来会越来越多地使用流，你会发现自己要调用接受 Stream 参数的方法。如果使用非线程安全的行为型参数创建管道，就像方才那样：

```
Stream<Book> books = library.stream()
.peek(/* non-threadsafe behavioral parameter */);
```

然后调用库方法并将 books 作为参数提供，那么库方法可能会在调用终止操作前对你的流调用 parallel，这会导致上面的竞态条件。类似于 sequential，方法 parallel 会给实现一个暗示，指出将要采用哪种执行模式；对于整个管道来说，执行模式只在终止操作开始执行时才确定下来，因此对其的控制并不属于作为管道创建者的你。这个示例告诉我们，使用非线程安全的行为型参数的流早晚会出问题。

本节介绍的分类 map 的示例可能会导致你认为有状态行为型参数的主要问题在于线程安全性。事实上，java.util.stream 的文档对有状态操作给出了一个一般性描述“结果依赖于状态，而状态会在流管道的执行过程中发生变化”。我们来看一个尽管是线程安全的，但有状态操作还是会出错的示例，我们让每个 Book 指向另外一个 Book，并且在 Book 中包含一个 boolean 字段，表示是否有其他 Book 指向自己。为了测试，我们创建一个 List<Book>，其中每个 Book 都指向列表中的相邻元素(将列表看作是一个环形)，然后执行如下代码：

```
long count = bookList().stream()
    .peek(b ->b.refersTo.referred = true)
    .filter(b ->b.referred) // stateful!
    .count();
```

如果顺序运行一个包含 1000 个元素的列表，那么结果会是 1 或 999，这取决于每个元素指向的是其前驱还是后继(这种差别完全取决于实现的计算顺序选择，请不要这么编写代码)。不过要是并行运行，那么结果就不一致了，可能是 1 和 4 之间、995 或是 999，完全无法预测。

对于并行代码来说，另一个同样重要的需求是管道在终止操作执行时要防止源发生变化。现在来考虑一下集合，防止源发生变化会提醒你不要在迭代时改变集合结构：如果检测到集合的结构发生了变化(除了自己做的修改)，那么“快速失败”的迭代器(由非线程安全的集合所创建)就会抛出 ConcurrentModificationException 异常。分割迭代器(参见 5.2 节)是迭代器的并行版本，如果它在终止操作执行时检测到流的源发生了结构上的变化，那么其行为会类似于迭代器。

这是并行流的源出现修改情况最为常见的一种，但绝非唯一一种：ConcurrentModificationException 异常只会因结构修改而抛出，一般是向集合添加元素或是从集合中删除元素。不过对于流来说，规则会禁止对流的源进行任何修改，包括改变元素的值，不仅仅是管道操作，任何线程都不可以这么做(唯一一个例外就是线程安全的并发数据结构，例如 java.util.concurrent 包中的)。这个限制其实并没有那么严格，考虑这种可能：在非线程安全的数据结构上执行并行操作。

3.2.4 终止管道

上一节的诸多示例有一个共同点：由于这些示例所演示的流操作都是延迟计算的，因此其效果就是将各个流组合到管道中而不会对任何元素进行处理。与之相反，本节所介绍的操作都是立即执行的；对流调用其中任何一个方法都会开始流元素的计算，

将元素从流的源中拉取出来。由于所有这些操作的结果都不是流，因此从某种意义上来说，它们都是将流的内容汇聚为单个值。不过最好将其划分为 3 个类别：

- 搜索操作，用于检测满足某种约束条件的流元素，因此有时在没有处理完整个流时就会结束。
- 汇聚，会返回单个值，作为对流元素值的一个总结。这个主题有些大，下一章将会专门对其进行介绍。现在，我们只关注两个方面：便捷的汇聚(Reduction)方法，如 count 和 max，以及简单的收集器，它会通过将元素累加到集合中来终止流。
- 副作用操作，这个类别只包含了两个方法：forEach 与 forEachOrdered。Stream API 中只有这两个终止操作带有副作用。

搜索操作

可以划分为“搜索”操作的 Stream 方法分为两组：第 1 组包含了匹配操作，它们会测试是否有某个流元素或全部流元素满足给定的 Predicate：

Stream<T>	ⓘ
anyMatch(Predicate<T>)	boolean
allMatch(Predicate<T>)	boolean
noneMatch(Predicate<T>)	boolean

anyMatch 在找到与 predicate 匹配的元素时会返回 true；allMatch在找到不满足predicate的任意一个元素时会返回false，否则返回true；noneMatch与之类似，如果找到任意一个满足predicate的元素时会返回false，否则返回true。

例如，在规划书架的组织时会遇到一个难题，即这些书架的

高度都不同。如果要知道历史书是否可以放到书架上面(其净高只有 19 厘米)，那么可以编写下面的代码：

```
boolean withinShelfHeight = library.stream()
    .filter(b ->b.getTopic() == HISTORY)
    .allMatch(b ->b.getHeight() < 19);
```

不过，如果我规划不同主题的图书在书架上的摆放位置时，那么这就不是我想要的结果了。最好在一个单独的操作中能够确定可以放在这个矮书架上的所有主题的图书(更好的方式则是能够计算出每个主题的图书所需要的净高)。这个目标可以通过将上述代码包装到一个循环中来达成，不过代价却是要重复调用 library.stream——这个解决方案效率很低且丑陋不堪。第 4 章将会介绍使用收集器实现的更好的解决方案。

值得注意的是，与标准的逻辑规则一样，在空的流上调用 allMatch 总是返回 true。

第 2 组搜索操作由两个“find”方法构成：findFirst 与 findAny：

Stream<T>	ⓘ
findFirst()	Optional<T>
findAny()	Optional<T>

如果找到，那么这两个方法会返回一个流元素，区别就在于返回的元素可能是不同的。其返回类型需要解释一下。乍一看，我们可能会编写下面这样的代码：

```
Book anyBook = library.stream()
    .filter(b ->b.getAuthors().contains("Herman Melville"))
    .findAny(); // doesn't compile
```

如果 library 中没有元素会怎样呢，这样其所供给的流就为空？在这种情况下，findAny 到底返回什么是无从得知的。返回

null 这种传统的 Java 解决方案并不令人满意：其表示的结果是模糊的，到底是匹配了流中的 null 还是没有找到匹配的元素，就像 Map.get 的结果一样。此外，null 无法用于原生流。Java 8 所提供的解决方案就是类 java.util.Optional<T>；该类的实例是一个包装器，可能包含也可能不包含非 null 的 T 类型值。“find”方法考虑了空流的可能性，因此它返回的是 Optional，这样上述代码的正确版本应该如下所示：

```
Optional<Book> anyBook = library.stream()
    .filter(b ->b.getAuthors().contains("Herman Melville"))
    .findAny();
```

下一节将会简要介绍一下 Optional 类。

与 findAny 相反，假设流是有序的，findFirst 会返回遇到的第 1 个元素。例如，我们可以通过如下代码从本书文本中找到包含字符串“findFirst”的第 1 行：

```
    BufferedReader br = new BufferedReader(new
FileReader("Mastering.tex"));
    Optional<String> line = br.lines()
        .filter(s ->s.contains("findFirst"))
        .findFirst();
```

findFirst 的用例与此类似，其问题描述关注于在有序流中找到第 1 个匹配的元素。另一方面，如果有序流中的任何一个匹配的元素都可以接受，那么你就应该使用 findAny；findFirst 在维持顺序上要做一些不必要的工作，而这是我们不需要的。对于无序流来说，这两个方法之间并没有本质的差别。

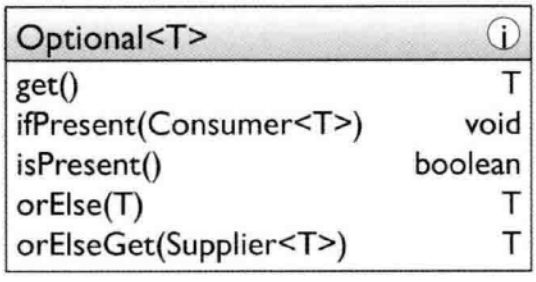
搜索操作搭配延迟计算会降低工作量，就像在本章一开始介绍的那样：找到满足 Predicate 的一个元素后匹配方法就可以返回了(也可能不返回，这取决于具体的匹配操作)。“find”方法总是

在找到一个元素后就返回。延迟计算确保在这些情况下，流操作会停止，并且不会生成不必要的元素(或者只生成极少量的元素)。

类 Optional

我们在上一节看到 Optional<T>用作 find 操作的返回值，之前(3.1.2 节)我们曾遇到过其原生类型变种 OptionalInt。在对空流应用终止操作时我们需要 Optional 及其变种，例如，当过滤器排除了所有元素时。上一节谈到了使用 null 的问题；Optional<T>通过提供一个永远不会与 T 混淆的特殊空值来避免这些问题。

对于本书中所使用的 Optional 场景来说，下面简要介绍一下它的几个方法，客户端可以通过这些方法获取其值：



这些操作的目的如下所示：

- get：如果存在则返回一个值；否则，该方法会抛出 NoSuchElementException 异常。这是个“不安全的”访问一个 Optional 内容的操作，通常情况下应该避免使用，转而使用如下安全的方法。
- ifPresent：如果值存在，那么将其提供给 Consumer；否则，什么都不做。
- isPresent：如果值存在，那么返回 true；否则返回 false。
- orElse：如果值存在，那么将其返回，否则返回参数。它与 orElseGet 是访问内容的安全操作。对于 Optional 的一

般使用场景来说，即便存在空值的可能，这些操作也要比 get 用处更大。

- orElseGet：如果值存在，那么将其返回；否则，调用 Supplier 并返回其结果。

汇聚方法

Stream API 的设计借鉴了 Larry Wall 的著名口号，即让简单的事情保持简单，同时让困难的事情成为可能。我们将会在接下来的两章中看到如何通过收集器与分割迭代器来完成复杂的工作。不过实际上，大多数工作都是简单的，对于这些简单的工作，我们有专门为简单工作设计的 Stream.reduce 变种(参见 4.4 节)。数字原生流(如 IntStream)提供了更多的特性：

IntStream	ⓘ
sum()	int
min()	OptionalInt
max()	OptionalInt
count()	long
average()	OptionalDouble
summaryStatistics()	IntSummaryStatistics

除了最后一个方法 summaryStatistics 外，其他方法都是见名知意的。最后一个方法会创建出类 IntSummaryStatistics 的一个实例，它是一个拥有 5 个属性的值对象：average、count、max、min 与 sum。如果想要对数据做一次遍历后得到多个结果，那么它就是非常有用的了。例如，我们可以编写下面这样的代码，获取并打印出图书馆中图书的页数统计信息(对多个卷的页数求和)：

```
IntSummaryStatistics pageCountStatistics = library.stream()
    .mapToInt(b -> IntStream.of(b.getPageCounts()).sum())
    .summaryStatistics();
```

```
System.out.println(pageCountStatistics);
```

这会产生如下输出：

```
IntSummaryStatistics{count=409, sum=93641, min=158,
average=228.9511, max=1472}
```

同样的模式也出现在 LongStream 与 DoubleStream 中。引用流也有一些便捷的汇聚方法：

Stream<T>	ⓘ
count()	long
min(Comparator<T>)	Optional<T>
max(Comparator<T>)	Optional<T>

例如，如下代码会找出图书馆中最早出版的图书：

```
Optional<Book> oldest = library.stream()
    .min(Comparator.comparing(Book::getPubDate));
```

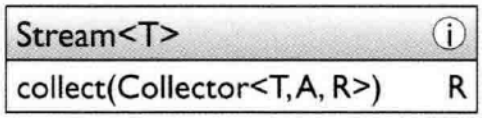
注意，为了找出拥有自然顺序的 Stream 元素中的最小值与最大值，你需要显式提供一个 Comparator。例如，为了找出按字母表排序的图书馆中的第 1 本书，可以编写如下代码：

```
Optional<String> firstTitle = library.stream()
    .map(Book::getTitle)
    .min(Comparator.naturalOrder());
```

类似的工厂类 Comparator.reverseOrder 则用于反转自然排序。

收集流元素

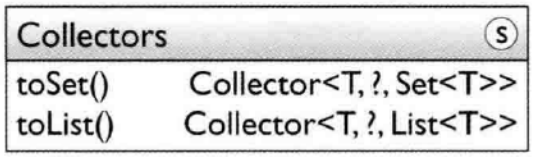
对于引用流来说，第 2 种汇聚使用 Stream.collect 将流值积聚到可变容器中，例如 Java 集合框架中的类，这称为可变汇聚：



collect 的参数是 Collector 接口的一个实例。如上图所示，Collector 有 3 个类型参数：第 1 个参数表示被收集的类型，第 3 个参数表示结果容器的类型；第 2 个参数将在 4.3.1 节介绍，作为深入探索收集器的一部分内容进行讲解。

大多数集合用例都可以通过类 Collectors 的工厂方法所返回的预定义 Collector 实现所涵盖。本章将会介绍其中最简单的 3 个方法；第 4 章将会对这些方法进行完整的介绍，如果需要的不属于预定义实现所能提供的，那么第 4 章还会介绍如何编写自己的。

这里所介绍的 3 个工厂方法会将流元素收集到 3 个主要的 Java 集合框架接口：Set、List 与 Map 其中一个实现中(具体由框架选择)。下面给出方法 toSet 与 toList 的声明。



例如，要想将图书馆中图书的标题收集到一个 Set 中，可以通过如下代码实现：

```
Set<String> titles = library.stream()
    .map(Book::getTitle)
    .collect(Collectors.toSet());
```

常见做法是静态导入 Collectors 工厂方法，这样最后一行代码就变成了下面这样：

```
.collect(toSet());
```

图 3-5 展示了收集器是如何处理这两本书的。在(a)中，输入元素是之前看到的两个 Book 实例，在(b)中，它们被放到了 Set 中；在内部，操作是通过方法 Set.add 实现的，同时语义保持不变：元素是无序的，重复元素会被丢弃。在(c)中，流被处理完毕，装配好的容器会从收集器返回。

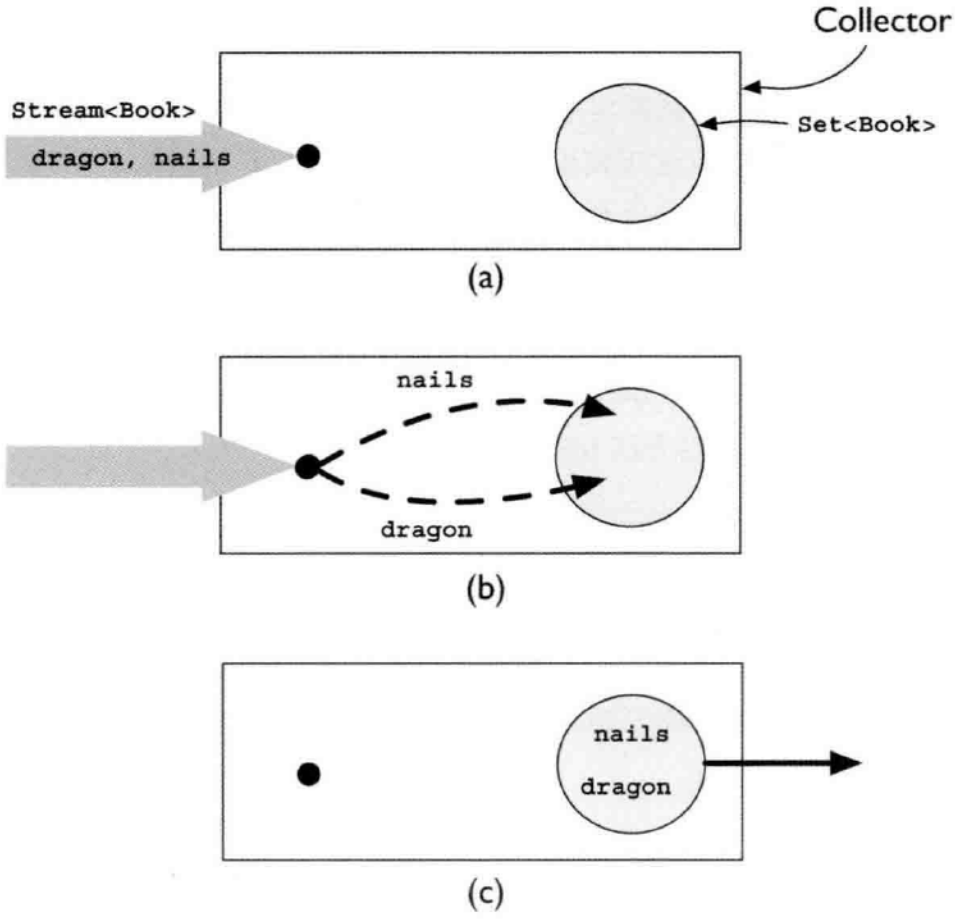


图 3-5 Collectors.toSet 的工作原理

方法 toList 非常类似于 toSet，通过方法 List.add 实现了元素的积聚；这样，如果流是有序的，那么所创建的 List 就会有相同的顺序(如果不是这样，那就说明流元素以不确定的顺序被添加进

来)。不过，创建一个会积聚为 Map 的收集器要稍微复杂一些。toMap 的两个重载方法就是完成这个目标的，每个方法都接受一个从 T 到 K 的键抽取函数以及从 T 到 U 的值抽取函数。这两个函数会应用到每个流元素上，从而生成一个键值对。

Collectors	Ⓢ
toMap(Function<T,K>, Function<T,U>)	Collector<T, ?, Map<K,U>>
toMap(Function<T,K>, Function<T,U>, BinaryOperator<U>)	Collector<T, ?, Map<K,U>>

例如，可以通过 toMap 的第 1 个重载收集器将集合中的每本图书的标题映射到其出版日期上：

```
Map<String,Year> titleToPubDate = library.stream()
    .collect(toMap(Book::getTitle, Book::getPubDate));
```

图3-6展示了toMap收集器的这两个函数装配Map的工作原理。

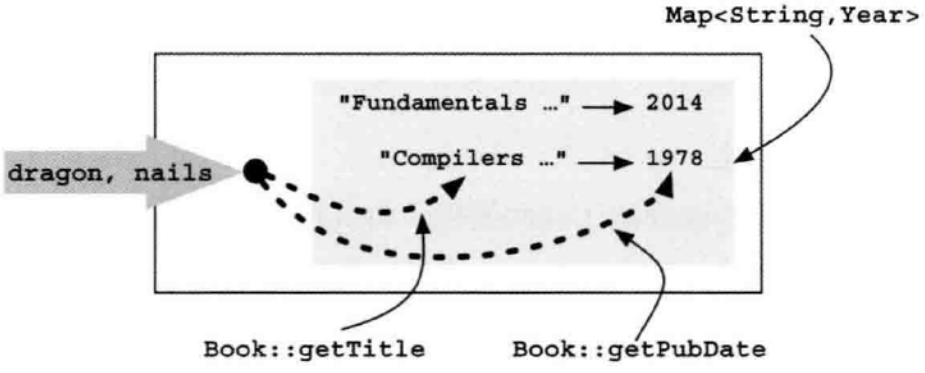


图 3-6　Collectors.toMap 的工作原理

toMap 的第 2 个重载方法考虑到了重复键的情况。如果很喜欢某本书，那么我就会在新版发布时将其购买回来。不同版本拥有相同的标题，但出版日期是不同的。当然了，Map 不能包含重复的键，因此对于上述代码来说，如果某本书有多个版本，那么收集器就会抛出 IllegalStateException 异常。toMap 的这个重载版

本就是为了解决重复键的问题。

如果需要重复键，那么就可以使用第 2 个重载方法，这样程序员就可以指定具体行为了，这是通过类型为 BinaryOperator<U> 的 merge 函数形式指定的，它会从两个已有值(一个位于 Map 中，另一个是要添加到 Map 中的)生成一个新值。有多种方式可以通过两个值来生成相同类型的第 3 个值；例如，两个 String 值可以拼接。这里，我决定只在 Map 中包含每本书的最新版本：

```
Map<String,Year> titleToPubDate = library.stream()
    .collect(toMap(Book::getTitle,
                   Book::getPubDate,
                   (x, y) -> x.isAfter(y) ? x : y));
```

由于 toMap 返回的收集器(就像 toSet 与 toList 返回的一样)会被积聚到非线程安全的容器中，因此管理多线程环境下结果的积聚会对并行流的性能造成影响。第 4 章将会介绍如何更高效地将结果积聚到 Map 中，第 6 章则会介绍收集器对流性能的影响。

副作用操作

本书一开始就介绍了这些操作，将其作为替代外部迭代的最简单的一种方式。它们会终止一个流，按照顺序对每个元素应用相同的 Consumer。Stream API 不支持有副作用的操作，但它们却是个例外：

Stream<T>	ⓘ
forEach(Consumer<T>)	void
forEachOrdered(Consumer<T>)	void

我们之前已经看到了这些方法的具体使用。它们之间的主要差别从名字就能看出来：forEach 用于在并行流上的高效执行，因

此它不保留顺序。另外一点不那么容易看出来，它并不保证操作的同步，操作可能会在不同的线程上执行。例如，假设我想要计算图书馆中全部图书的总页数，那么我可能幼稚地声明一个实例变量 pageCount，然后不断向其写入来计算出总页数，代码可能会像下面这样：

```
// don't do this - race conditions!
library.stream()
    .forEach(b -> {pageCount += b.getPageCount();});
```

上述代码是错误的，因为增加 pageCount 可能发生在没有同步的不同线程上，因此很容易出现竞态条件(同时干扰执行)。当然，即便流是串行的，但编写并行代码的原则告诉我不能这么写代码。forEach 的 API 文档还警告说动作可能“在库所选择的任何线程中”执行。

可以通过 forEachOrdered 来“纠正”上述代码，相对于 forEach，它会保留顺序并确保同步。下面代码的结果就是正确的：

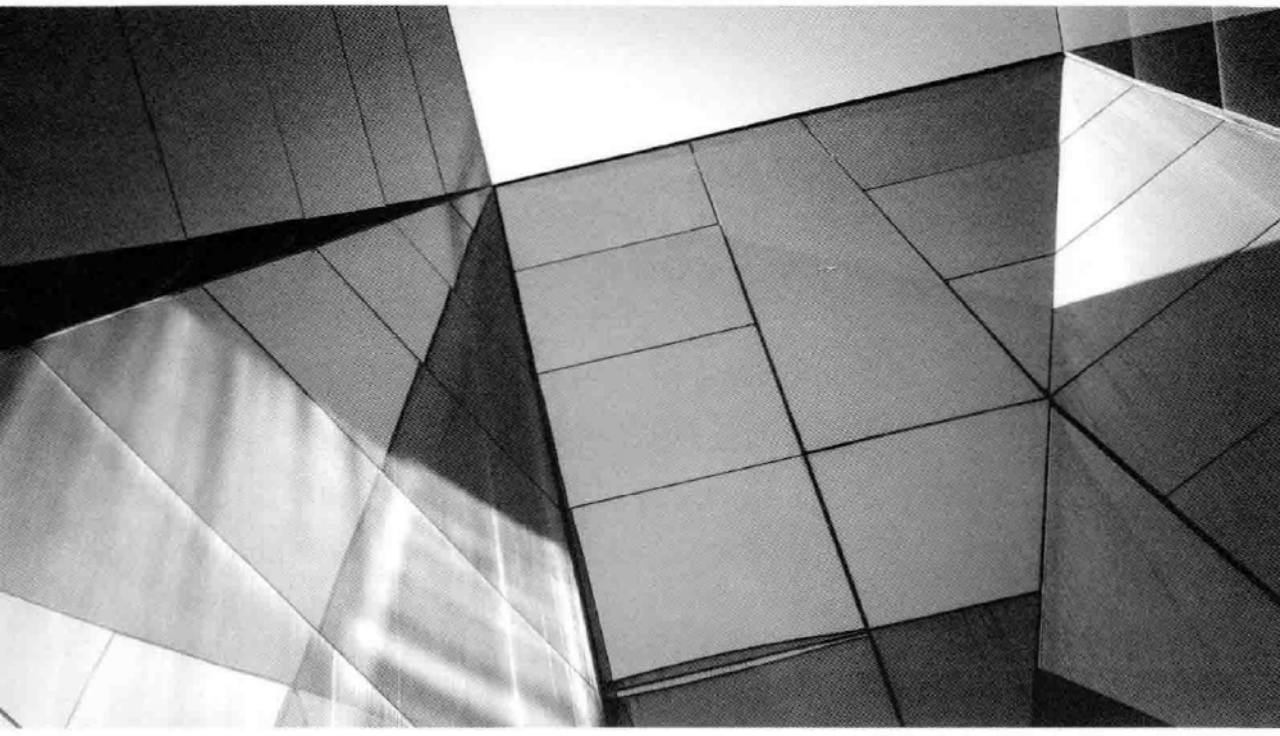
```
// formally correct but inefficient and ugly
library.stream()
    .forEachOrdered(b -> {pageCount += b.getPageCount();});
```

虽然代码可以使用，但它却不是面向并行的：forEachOrdered 的使用会强制在串行模式下执行。通常来说，遵循 Stream API 的代码既要保持可读性，又要确保高效率。

```
int totalPageCount = library.stream()
    .mapToInt(Book::getPageCount)
    .sum();
```

3.3 小结

现在我们已经对基本的流与管道操作有了总体的认识，不过还需要通过一些使用这种编程风格来解决实际问题的示例来加深理解。接下来的两章将立足于我们已经学到的知识，更深入地探索 API，并探究将对流的复杂查询分解为按步操作的各种策略。



第4章

终止流：收集与汇聚

管道是由终止操作结束的，第3章曾介绍过，终止操作分为3组：搜索操作、汇聚以及带有副作用的操作。虽然已经介绍过了这3组终止操作，但关于汇聚还有很多内容值得探讨，本章就将完成这个任务。

广义上来说，所谓汇聚指的是返回单个值的操作，它以某种方式总结了流元素的值。不过，这个说明并未对会创建新对象的操作与改变其操作数的操作做出区分。而这对于Java程序员来说

又是非常重要的。关于汇聚的传统思想已经付诸于那些支持不变性的语言了；虽然现代 Java 编程实践一般来说会鼓励不变性，但大多数管道还是以可变收集作为结束。因此，收集(也称为可变汇聚)是 Stream API 中最为重要的终止操作。收集是传统汇聚的广义化，在 Java 编程中主要对于原生流会起到很大的帮助作用。收集也会将流元素中的值总结到单个对象中，从这一点看收集类似于汇聚，不过它是通过变化来实现的。

下面对传统的 Java 批处理代码与收集做一个简单的比较。如果将实际的数据源建模为 Iterable<Book>(这里声明为 library)，那么我们通常可以将它里面的值积聚到 List<Book>中，就像下面这样：

```
List<Book> bookList = new ArrayList<>();
for (Book b : library ) {
   bookList.add(b);
}
```

在 Java 8 中，可以通过 Stream<Book>对数据源建模，因此如下代码的效果与上面是一样的：

```
List<Book> bookList = libraryStream
.collect(Collectors.toList());
```

除了显而易见的在简洁性与可读性上的改进外，这个收集器版本还具有很多优势：即便不断积聚流元素的 List(当前实现中使用的是 ArrayList)不是线程安全的，流操作还是能够安全地以并行方式执行。此外，收集器模式是非常灵活的，并且可以轻松地进行组合：图 4-1 概览了本章后面将要介绍的一个示例，让大家提前感受一下这种灵活性。

收集器模式的示例

根据主题对图书进行分类的 Map：

```
Map<Topic,List<Book>> booksByTopic = library.stream()
    .collect(groupingBy(Book::getTopic));
```

从图书标题映射到最新版发布日期的有序 Map：

```
Map<String,Year> titleToPubDate = library.stream()
    .collect(toMap(Book::getTitle,
                   Book::getPubDate,
                   BinaryOperator.maxBy(naturalOrder()),
                   TreeMap::new));
```

将图书划分为小说(对应 true)与非小说(对应 false)的 Map：

```
Map<Boolean,List<Book>> fictionOrNon = library.stream()
    .collect(partitioningBy(b -> b.getTopic() == FICTION));
```

将每个主题关联到该主题下拥有最多作者的图书上：

```
Map<Topic,Optional<Book>> mostAuthorsByTopic =
library.stream()
    .collect(groupingBy(Book::getTopic,
       maxBy(comparing(b -> b.getAuthors().size()))));
```

将每个主题关联到该主题总的卷数上：

```
Map<Topic,Integer> volumeCountByTopic = library.stream()
    .collect(groupingBy(Book::getTopic,
       summingInt(b -> b.getPageCounts().length)));
```

拥有最多图书的主题：

```
Optional<Topic> mostPopularTopic = library.stream()
.collect(groupingBy(Book::getTopic, counting()))
.entrySet().stream()
.max(Map.Entry.comparingByValue())
.map(Map.Entry::getKey);
```

将每个主题关联到该主题下所有图书标题拼接成的字符串上：

```
Map<Topic,String> concatenatedTitlesByTopic = library.stream()
.collect(groupingBy(Book::getTopic,
mapping(Book::getTitle, joining(";"))));
```

图 4-1　Stream 操作的示例

在这个命令式版本中，关键部分是容器的创建(new ArrayList<>())以及将元素积聚到其中的操作(bookList.add(b))。收集器也提供了完成这两个任务的组件，分别称为提供器与积聚器。我们可以在图中识别出第 3 章介绍的这些组件。图 4-2 展示了图 3-5(Collectors.toSet 所创建的收集器)与这两个组件之间的关系。

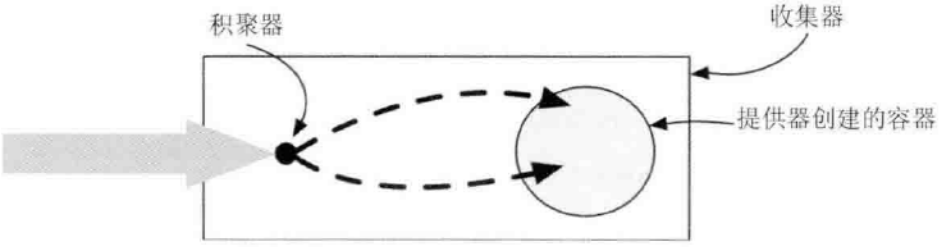


图 4-2　Collector.toSet 的组件

提供器定义了容器类型，在图中显示为阴影(在图 4-2 中就是框架选择的 Set<Book>实现)部分。积聚器会根据进来的流装配容器；在图 4-2 中就是 Set::add。

4.1　使用收集器

本章的后续章节会深入探索收集器模式的想法，同时会介绍何时以及如何创建自定义收集器。不过在这之前，我们应该考虑其最常见以及最直接的使用模式：即 Collectors 类的工厂方法所提供的预定义收集器的使用。可以将其分为两类：独立收集器(单独使用)以及与其他收集器组合使用的收集器。

4.1.1　独立的预定义收集器

根据功能，独立收集器可以分为 3 组：

- 积聚到框架提供的容器中。

- 积聚到自定义的集合中。
- 将元素积聚到分类 Map 中。

我们已经在第 3 章见过了第 1 组中的大部分——提供者是框架提供的集合：对于 toSet 来说就是 Set 的一个实现，对于 toList 来说就是 List 的一个实现，对于 toMap 来说就是 Map 的一个实现。这些收集器的积聚器也是显而易见的：对于 Collection 实现就是 add，对于 toMap 收集器来说就是 put。

该组中还有一个 joining 方法，它返回的收集器会将 String 对象流连接到 StringBuilder 中，接下来其内容会以 String 形式返回(4.3.1 节将会对此做进一步的介绍)：

Collectors	Ⓢ
joining()	Collector<CharSequence,?,String>
joining(CharSequence)	Collector<CharSequence,?,String>
joining(CharSequence, CharSequence, CharSequence)	Collector<CharSequence,?,String>

joining 的第 1 个重载方法只是简单地连接输入流中的字符串。第 2 个重载方法接受一个 CharSequence，并将其作为分隔符插入到输入字符串中。例如，如下代码会将图书馆中所有图书的标题拼接起来，并通过一个双冒号对其进行分隔：

```
String concatenatedTitles = library.stream()
    .map(Book::getTitle)
    .collect(joining("::"));
```

第 3 个重载方法接受一个分隔符、一个前缀和一个后缀；例如，对于一本书 b 来说，如下代码会生成一个拼接该书作者的字符串，中间通过逗号分隔，该字符串以图书的标题开始，以一个换行符结束：

```
b.getAuthors().stream().collect(joining(
        ",",
```

```
        b.getTitle() +": ",
        "\\n"))
```

我们可以通过如下代码创建一个字符串列表，其中每个字符串都包含了一本书的所有作者名：

```
List<String> authorsForBooks = library.stream()
    .map(b -> b.getAuthors().stream()
        .collect(joining(", ", b.getTitle() + ": ", "")))
        .collect(toList());
```

到目前为止，我们见到的所有收集器都会将元素积聚到框架选择的集合中。很多 Collectors 方法都有一些变种，可以指定容器的提供者：

Collectors	Ⓢ
<M extends Map<K,U>> toMap(Function<T,K> keyExtracter, Function<T,U> valueExtracter, BinaryOperator<U> mergeFunction, Supplier<M> mapFactory) Collector<T,?,M>	
<C extends Collection<T>> toCollection(Supplier<C> collectionFactory) Collector<T,?,C>	

例如，我们之间看到合并功能中使用的 toMap 就是用于构建一个从图书标题映射到最新版日期的 Map。如果随后要以标题的字母表顺序显示 Map 的内容，那么将其放到有序 Map 中就会改进性能：

```
Map<String,Year> titleToPubDate = library.stream()
.collect(toMap(Book::getTitle,
Book::getPubDate,                                    ❶
(x, y) -> x.isAfter(y) ? x : y,
TreeMap::new));
```

由于合并功能实际上是使用自然顺序选择两个 java.time.Year 值中较大的一个，因此代码行❶可以替换为如下代码：

```
BinaryOperator.maxBy(Comparator.naturalOrder());
```

我们看到的3个toMap重载方法的通用性越来越高：第1个只接受键值抽取函数；第2个还接受一个合并函数；第3个则又接受一个提供器。这意味着，如果想要指定一个提供器，那么你就需要提供一个合并函数。不过，如果打算这么做，那么你可以重新创建简单版本的toMap的行为，允许出现重复的键，只需要将

```
(x,y) -> { throw new IllegalStateException(); }
```

作为该重载方法的第 3 个参数。

你可能希望也会有相应的 toList 与 toSet 的重载方法，它们创建的收集器可以指定自定义的提供器。但实际上，相对于为 toList 与 toSet 增加额外的重载方法，现在已经有了一个更为通用的 toCollection 方法。它更加通用：通过它不仅可以选择 Set 与 List 的任意实现，还可以选择 Collection 的任意子接口。例如，我们可以将流元素收集到一个有序集合中或是阻塞队列中：

```
NavigableSet<String> sortedTitles = library.stream()
    .map(Book::getTitle)
    .collect(toCollection(TreeSet::new));
BlockingQueue<Book> queueInPubDateOrder = library.stream()
    .sorted(Comparator.comparing(Book::getPubDate))
    .collect(toCollection(LinkedBlockingQueue::new));
```

第 3 组 Collectors 方法会返回带有流元素分类功能的收集器。它们与 toMap 返回的收集器相关，只不过相对于使用值抽取函数来说，它们放到 Map 中的值是元素本身，或是元素列表，一个 List 对应于每个分类键：

Collectors	Ⓢ
groupingBy(Function<T,K>)	Collector<T,?,Map<K,List<T>>>

例如，可以像下面这样根据主题对图书进行分类：

```
Map<Topic,List<Book>> booksByTopic = library.stream()
    .collect(groupingBy(Book::getTopic));
```

图 4-3 展示了该示例的工作方式：对于每个输入的元素来说，分类函数都会被调用以确定其键。如果键不存在，那么它就会被添加进来，其值则是个只包含当前元素的单例 List。如果键与 List 值已经存在，那么流元素就会被添加到列表中。该收集器的提供器是框架选择的 Map<Topic, List<Book>>实现的无参构造器，积聚器具有之前描述的添加每一个输入的元素的功能。这些默认情况都可以被重写，下一节将会对此进行介绍。

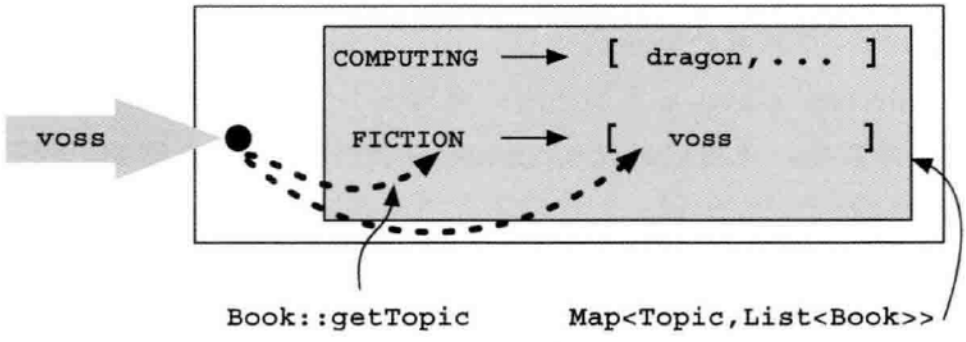


图 4-3 Collectors.groupingBy 的工作原理

groupingBy 的一个变种是便捷方法 partitioningBy，其中键类型 K 被指定为 Boolean：

Collectors	Ⓢ
partitioningBy(Predicate<T>)	Collector<T,?,Map<Boolean,List<T>>>

例如，如下代码会将 true 映射为小说列表，将 false 映射为非

小说列表：

```
Map<Boolean,List<Book>> fictionOrNonFiction = library.stream()
    .collect(partitioningBy(b -> b.getTopic() == FICTION ||
                            b.getTopic() == SCIENCE_FICTION));
```

图 4-4 展示了该示例。

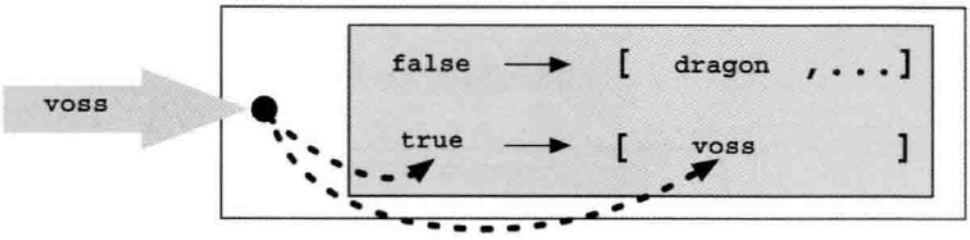


图 4-4　Collectors.partitioningBy 的工作原理

4.1.2　组合收集器

独立收集器本身是很有用的，不过 Collector API 真正强大之处在于收集器可以协同工作，并与其他功能搭配使用。在 Java 8 的设计中，组合的重要性在第 3 章一开始就曾介绍过：Collector API 就是这种设计原则的一个佐证。

举个例子，我想表示一组值的分布情况，例如图书馆中的图书根据每个主题的分布情况。上一节介绍过分类收集器，它可以生成一组映射，其中键是属性，例如图书的主题。不过，这些映射的值类型已经确定为是流元素的一个列表。要想表示分布情况，该列表需要被替换为元素数量。这只是使用值替换映射中的元素列表的一个示例：还可以使用元素衍生属性的列表(例如图书的发布者)，或是其他针对元素值的汇聚(时间最为久远的主题，页数总和等)。相对于为这么多用例提供专门的分类收集器，Collector API 则提供了一个扩展点，可以将收集器组合起来。

通过组合，我们可以将多个收集器或其他操作的结果组合起

来创建新的收集器。其中最为重要的一种形式就是将groupingBy与第2个“下游”收集器组合起来。在该组合中，groupingBy提供了分类功能与分类键，与给定键相关的元素则被发送给下游收集器进行下一步处理。图4-5展示了它是如何处理我们之前见过的简单groupingBy重载的，其中默认的下游收集器就是Collectors.toList所返回的。

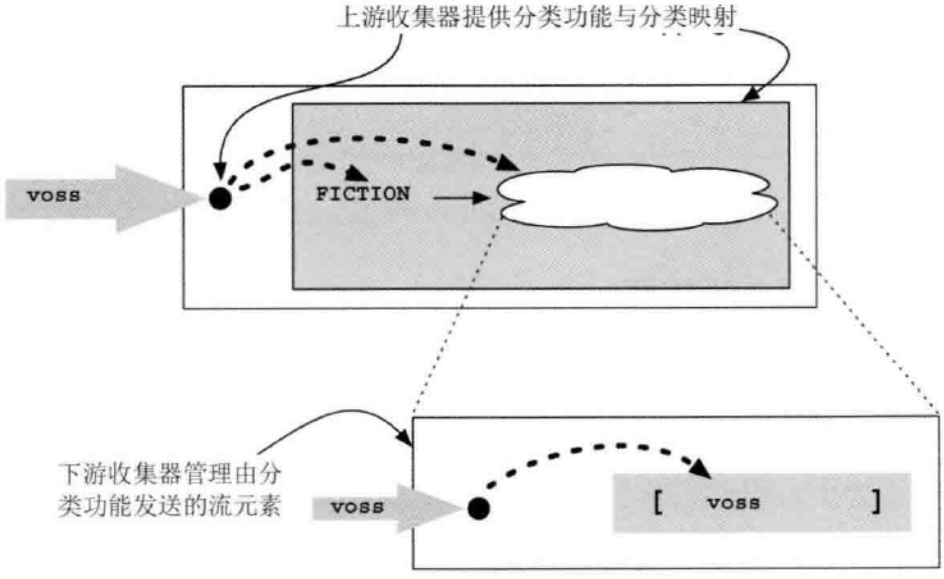


图 4-5 groupingBy 是一个组合

此外，还有一个 Collectors 工厂方法，它可以创建分类收集器与用户提供的下游收集器的组合：

Collectors	Ⓢ
groupingBy(Function<T,K>, Collector<T,A,D>)	Collector<T,?,Map<K,D>>

到目前为止，我们所看到的 groupingBy 收集器的行为相当于使用如下重载：

```
Map<Topic,List<Book>> booksByTopic = library.stream()
   .collect(groupingBy(Book::getTopic, Collectors.toList()));
```

回到分布表示这个示例，我们看到问题可以通过将groupingBy与不同的下游收集器组合起来而得到解决，这个下游收集器会计算输入的元素数量。事实上，有一个 Collectors 工厂方法是专门用于这个目的的：Collectors.counting(4.4.3 节将会介绍其实现)。图 4-6 展示了其在这个示例中的使用。

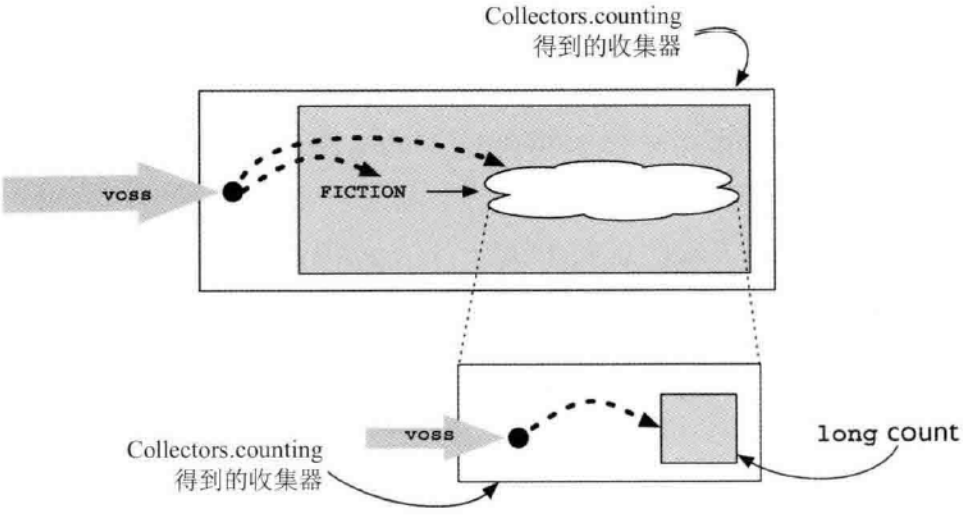


图 4-6　下游计算

为了表示分布，我们可以写成下面这样：

```
Map<Topic,Long> distributionByTopic = library.stream()
    .collect(groupingBy(Book::getTopic,
    Collectors.counting()));
```

其他收集器也可以用作 groupingBy 的下游。事实上，有很多 Collectors 工厂方法都是针对这个目的的。我们可以通过便捷汇聚的二元性将其与4.1.1节介绍的独立收集器区分开来；例如，counting返回的收集器是终止操作 count，它可以用作下游收集器。还例如：

- 对应于终止操作max与min的是Collectors工厂方法maxBy与minBy。例如，我们可以创建一个映射，其中包含每个主题下作者最多的图书：

```
Map<Topic,Optional<Book>> mostAuthorsByTopic =
library.stream()
      .collect(groupingBy(Book::getTopic,
      maxBy(comparing(b -> b.getAuthors().size()))));
```

- 对应于原生流终止操作 sum 与 average 的是 summingInt、summingLong 与 summingDouble 及其平均数版本返回的收集器。例如，我们可以创建一个映射，其中包含每一个主题下卷的总数(回忆一下，getPageCounts 返回的是一个 int 值数组，数组长度等于该书的总卷数)：

```
Map<Topic,Integer> volumeCountByTopic = library.stream()
   .collect(groupingBy(Book::getTopic,
   summingInt(b -> b.getPageCounts().length)));
```

还可以根据主题创建一个包含着图书平均厚度的 Map：

```
Map<Topic,Double> averageHeightByTopic = library.stream()
   .collect(groupingBy(Book::getTopic,
      averagingDouble(Book::getHeight)));
```

- 对应于终止操作 summaryStatistics 的是 summarizingInt、summarizingLong 与 summarizingDouble 返回的收集器。它们会将原生值收集到相应的 SummaryStatistics 对象中。例如，如下代码会生成针对卷数流的一个 IntSummaryStatistics 实例(而非像上一个示例那样的根据主题得到的图书卷数)：

```
Map<Topic,IntSummaryStatistics> volumeStats = library.stream()
   .collect(groupingBy(Book::getTopic,
```

```
        summarizingInt(b -> b.getPageCounts().length)));
```

可以通过如下代码打印：

```
System.out.println(volumeStats.get(Topic.COMPUTING));
```

得到下面这样的结果：

```
IntSummaryStatistics{count=94, sum=98, min=1,
average=1.042553, max=2}
```

- 对应于终止操作 reduce 的 3 个重载方法的是 reducing 的 3 个重载方法返回的收集器。我们将在本章结尾探索汇聚这个主题时再来对其进行介绍。

上述示例存在一些共同点：下游收集器(无论是统计元素数量、根据某个属性排序，还是对某个属性做统计)都会接收并处理整个流元素。不过，有时下游收集器只需要处理一个属性；例如，我想要创建一个从每个主题到该主题下所有标题字符串拼接的映射。从每本书中抽取出标题这个操作非常类似于对流的 map 操作，不过对于该示例来说，要在下游收集器接收到之前应用到 groupingBy 所分发的流对象上。这个需求可以通过 Collectors.mapping 所创建的收集器来实现，用户可以提供映射与下游收集器：

Collectors	Ⓢ
mapping(Function<T,U>, Collector<U,A,R>)	Collector<T,?,R>

图 4-7 展示了该方法所创建的收集器的工作方式。借助于 Collectors.mapping 的帮助，创建一个从标题到该主题下所有标题的拼接字符串的映射是很直接的：

```
Map<Topic,String> concatenatedTitlesByTopic = library.stream()
```

```
.collect(groupingBy(Book::getTopic,
        mapping(Book::getTitle, joining(";"))));
```

作为一种双重 mapping(在收集前会将一个函数应用到输入的值上)，collectingAndThen 接受一个函数，该函数会在收集后应用到容器上。“完成”操作将会在 4.3.1 节中介绍。

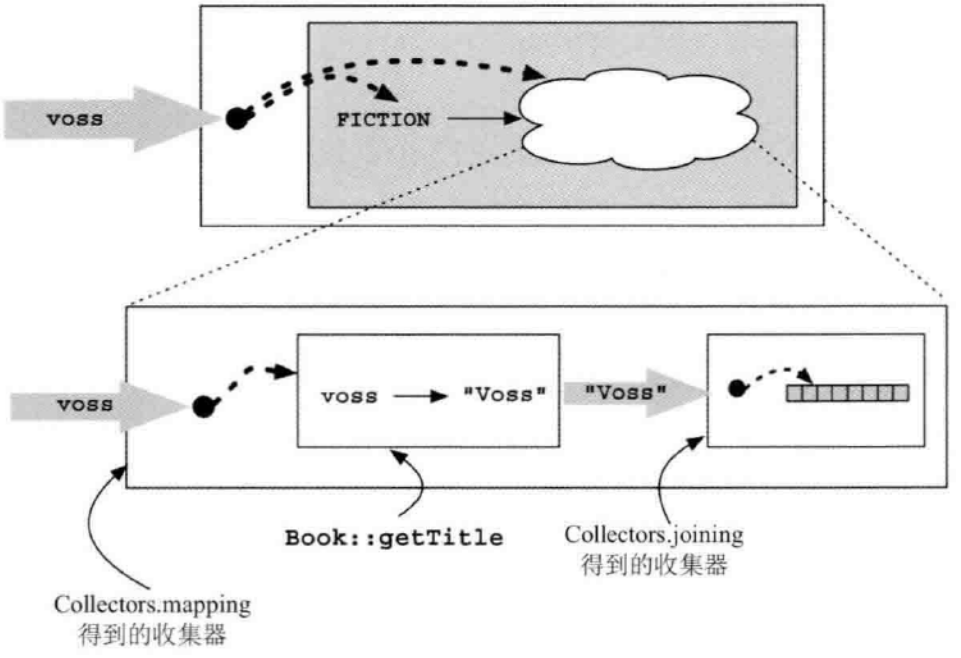


图 4-7　groupingBy 与 mapping 的组合

4.1.3　链接管道

上一节所介绍的技术提供了不同方式来处理针对每一个分类键的值——找到最大值、拼接字符串属性的值等。有些问题还需要更进一步的处理，其中不同键的值会一起处理。例如，就拿找到图书馆中最受欢迎的主题这一问题来说(即图书数量最多的主题)。我们知道如何生成一个从主题到图书数量的映射：

```
Map<Topic,Long> bookCountByTopic = library.stream()
.collect(groupingBy(Book::getTopic, counting()));
```

不过，现在 Map<Topic, Long>中的每个条目都需要进行比较才能找到数目最大的那个。在比较之前，图书馆中的所有元素都要经由收集器进行处理来创建分类映射；接下来才能比较主题、数量对，从而找到数值最大的那个。要想将 Stream API 的操作应用到下一阶段的处理，我们需要开启另一个流，它将这些配对作为自己的元素：

```
Stream<Map.Entry<Topic,Long>> entries = library.stream()
    .collect(groupingBy(Book::getTopic, counting()))
    .entrySet().stream();
```

我们现在可以通过方法Map.Entry.getKey与Map.Entry.getValue处理每个Map.Entry对象的组件了。Java 8增加了用于这种情况处理的比较器生成方法comparingByKey与comparingByValue。对于找到最流行的主题这一问题，我们可以通过如下代码实现：

```
Optional<Topic> mostPopularTopic = entries
    .max(Map.Entry.comparingByValue())
    .map(Map.Entry::getKey);
```

可以通过将两个管道链接起来模仿流式管道处理：

```
Optional<Topic> mostPopularTopic = library.stream()
    .collect(groupingBy(Book::getTopic, counting()))
    .entrySet().stream()
    .max(Map.Entry.comparingByValue())
    .map(Map.Entry::getKey);
```

对于 Java 8 的早期阶段来说，我们没办法预测惯用方式到底会如何演化。这么做的优缺点如下所示：

- 简洁，可读性好。
- 如果没有注意到优化只能应用到单独的两个管道上，以及需要足够的内存来容纳中间集合这一事实，那么这么做

就具有迷惑性。这样，它与某些中间操作的行为(例如sorted，在进入到下一步之前需要在内部收集流的整个内容)就没有什么差别了，第 6 章将会深入介绍该主题。

尽管存在性能问题，但这么做的目的取决于其对代码可读性与可维护性的整体影响。因此，本书还会继续使用它。

4.1.4 示例说明：最流行的主题

上一节最后，我们看到如何找到最流行的主题。不过代码并没有考虑到有多个主题拥有数量最多的图书的情况，如果是这样，那么我们想知道所有这些主题。本节将会给出该问题的解决方案；相对于简单地给出最终答案，我们会分步骤介绍。在每个阶段中都会停下来并思考该如何继续。首先来介绍一下这个问题的整体解决方案。

思考这类问题的一种方式是从目标往回推。新程序最重要的步骤也是使用max，就像之前一样，不过结果会包含所有最流行的主题而非单独一个。这表明它们需要放到集合中，如果这个问题类似于前面那个，那么我们只需要从map条目中找到最大的那一个即可，因此需要创建一个Map<Long, Set<Topic>>(Set或是List也可以)，其中键是每个主题的流行度，值是具有该流行度的主题集合。根据这一点，我们需要一个下面这样的Map，称为targetMap：

```
{98=[COMPUTING, FICTION], 33=[HISTORY]}
```

首先看看前两行代码生成的 Map：

```
{COMPUTING=98, FICTION=98, HISTORY=33}
```

将其称作 startMap。如何从 startMap 转换为 targetMap 呢？如果找不到解决方案，那么先停下来设计一个。提示：最简单的方法是使用 groupingBy。

如果 groupingBy 操作的目标是 targetMap，那么分类函数所抽取的键就一定要是流行度，由于它们是位于 startMap 中的值，因此分类函数就必须是 Map.Entry.getValue。

targetMap 中的值来自于 startMap 条目的键，因此 groupingBy 收集器下游的动作就一定是第 1 个，从输入的条目中抽取键，第 2 个动作则是将其积聚到 Set 中。这表明，我们需要将 groupingBy 与 mapping 收集器组合起来抽取键，收集器本身会与 toSet 收集器组合起来积聚它们：

```
startMap.entrySet().stream()
    .collect(groupingBy(Map.Entry::getValue,
          mapping(Map.Entry::getKey, toSet())));
```

将其放到一起，寻找最流行主题的代码如下所示：

```
Optional<Set<Topic>> mostPopularTopics = library.stream()
    .collect(groupingBy(Book::getTopic, counting()))
    .entrySet().stream()
    .collect(groupingBy(Map.Entry::getValue,
          mapping(Map.Entry::getKey, toSet())))
    .entrySet().stream()
    .max(Map.Entry.comparingByKey())
    .map(Map.Entry::getValue);
```

这并非问题的唯一解决方案。还可以考虑下面这些：

- 使用 Topic 作为排序键将 Book 元素插入有序 Map 中，并检索出元素的一个初始子集。
- 创建一个频率 Map 来确定最大值，然后积聚与该值关联的主题。
- 定义一个值对象(后面将会介绍)，然后直接对其进行汇聚。

如果花点时间研究一下这些解决方案，你会发现这种处理方式存在如此多的策略。

4.2 剖析收集器

正如第 1 章所述，递归解耦与集合之间存在直接的联系。回忆一下图 1-2：该图的合并阶段展示了如何通过两个 int 值生成 1 个新值，该示例中是得到两个值中较大的那个。这种汇聚适合于通过组合生成新值的算法，而非改变已有值，不过要想收集到可变容器中则需要对其做修改。图 4-8 展示了该收集器代码的一种可能的执行情况：

```
Stream.of(nails, dragon, voss)
    .collect(Collectors.toList());
```

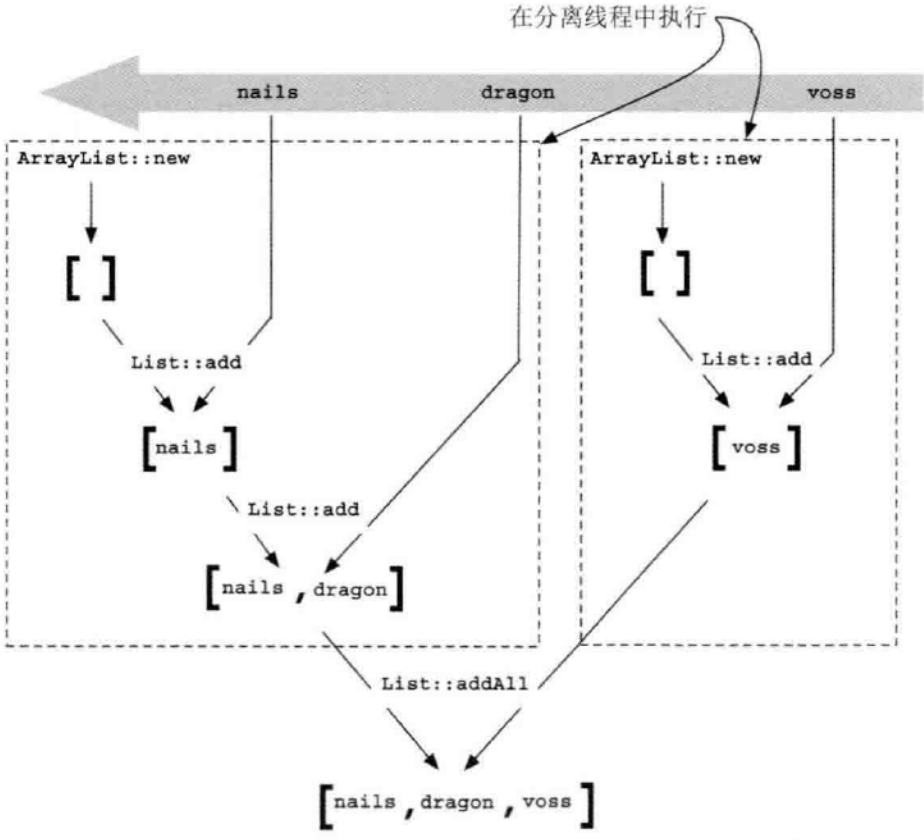


图 4-8　并行可变的汇聚：toList

图中表示的流元素按照从右向左的顺序排列，这样它们就与出现在源中的顺序保持一致了。不过，如果流是有序的，那么重要的就是它们在被积聚到结果容器的过程中其相对位置是保持不变的。

将值积聚到集合中的传统方式是创建一个新的，然后将元素逐个添加进去。图中表示的算法将这个串行过程变成了并行执行。对于每个线程来说，一个提供器函数会创建一个新容器(通常是个集合，不过一般来说，能积聚元素引用或元素值就可以，例如StringBuilder)。

接下来，每个线程都可以将元素积聚到容器中，它会使用一个积聚函数。最后，线程所创建的中间容器必须要合并起来；这需要第 3 个函数，即合并器，它用于将并行的连续积聚结果合并到一起。我们现在无须了解什么是合并器，因为合并器不会改变积聚器所定义的容器的功能性行为(4.3.3 节将会对此进行介绍)。

如果将流元素的类型看作 T(这里就是 Book)，将结果容器的类型看作 R(这里就是 List<Book>)，那么这些函数就是 2.4 节介绍的 3 个接口的实现：

- 用于创建新容器的函数是 ArrayList::new(Collectors.toList 的契约并不保证创建的 List 类型，不过在 Java 8 中它实际上是 ArrayList)；一般来说它是 Supplier<R>的一个实现。
- 用于向已有容器添加单个元素的函数是 List::add；一般来说它是 BiConsumer<R, T>的一个实现。
- 用于将现有容器组合起来的函数是 List::addAll；一般来说它是 BinaryOperator<R>的一个实现。

这是一个收集器的 3 个主要构成。它们要共同协作才能得到一致的结果，例如，我们很容易就能从图 4-8 中看到，它只不过是代码的众多可能执行方式中的一种，所有执行一定会生成相同的值。4.3.3 节介绍了确保自定义收集器能够做到这一点的规则，就像预定义的收集器所做的那样。

并发收集

我们可以从图 4-8 中清楚地看到，并行流处理中的性能瓶颈很有可能是将每个线程处理结果集合起来的组合操作。框架可以确保(假设收集器遵循着 4.3.3 节给出的规则)，给定部分上的积聚操作只会被单个线程所执行，同时合并器操作会安全执行，即便合并的是非线程安全的容器亦如此。不过显而易见的是，这种安全性需要付出性能的代价，特别是对于常见的 Map 合并操作。因此，框架提供了另外一种选择：并发收集器。

并发收集器指的是具备CONCURRENT特性的收集器(参见6.8节)。该声明告诉框架它可以从多个线程中对相同的结果容器调用积聚器函数。这会改进并发性能，代价则是丧失了流元素的顺序。因此，对于并发收集来说，最常见的用例就是收集到Map中，而且Collectors类中的每个toMap方法都有一个相应的toConcurrentMap方法，它会生成一个针对ConcurrentMap的收集器。与之类似，groupingBy的每个重载方法都有一个对应的groupingByConcurrent方法，它会返回一个ConcurrentMap而非Map。6.8节将会介绍这些方法所能带来的性能改进(以及应该在什么情况下使用它们)。

4.3　编写收集器

我们已经见识到了库所提供的预定义收集器的诸多功能，同时还看到了如何将其组合起来扩展功能。既然已经有了如此之多的功能，那为何还要定义自己的收集器呢？动机有二：你需要将值积聚到并未实现 Collection 的容器中(因此无法使用 Collectors.toCollection)，以及积聚的过程需要在被收集的值之间共享状态。

对于第 2 种情况，我们举个简单的示例，假设有一些根据某个属性排序的值，我们想要根据该属性的临近度对其进行分组。电力传输线的路线规划就是个很实际的示例，其中塔间距至多是一个常量(例如 MAX_DISTANCE)。包含一系列塔站的路线可以通过将它们分组为段来进行分析，在每一段中，塔与塔之间的距离都不能超过 MAX_DISTANCE，而不同段的两个塔之间的距离则要大于 MAX_DISTANCE。对于图 4-9 所示的简单示例来说，沿 X 轴规划的路线可能包含了如下点：

```
(3,0), (6,0), (8,0), (10,0), (14,0)
```

如果 MAX_DISTANCE == 2，那么这些点就可以分为 3 段：

```
[(3.0)], [(6,0), (8,0), (10,0)], [(14,0)]
```

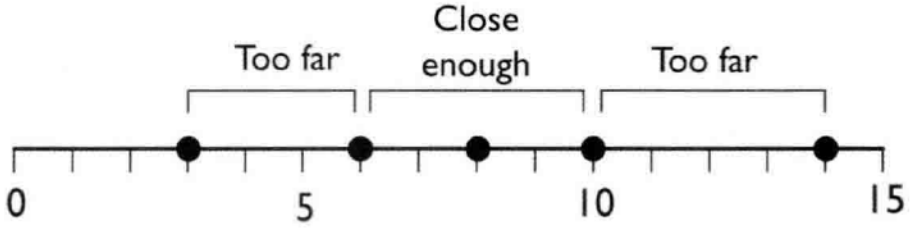


图 4-9　输电塔的位置

对于该问题来说，显而易见的表示形式就是一个嵌套的线性数据结构。由于向其中添加新元素需要访问最后一个元素，因此使用 Deque 而非 List 将会更加方便；此外，Deque 还有一个现代化且高效的实现 ArrayDeque。

下面是个简单的迭代式解决方案：

```
    Deque<Deque<Point>> groupByProximity(List<Point>
sortedPointList) {
    Deque<Deque<Point>> points = new ArrayDeque<>();
    points.add(new ArrayDeque<>());
    for (Point p : sortedPointList) {
       Deque<Point> lastSegment = points.getLast();
       if (! lastSegment.isEmpty() &&
             lastSegment.getLast().distance(p) > MAX_DISTANCE ) {
          Deque<Point> newSegment = new ArrayDeque<>();
          newSegment.add(p);
          points.add(newSegment);
       } else {
          lastSegment.add(p);
       }
       }
       return points;
    }
```

对于收集器实现来说，这是个很好的开始——实际上，只需做很少的修改就可以将其作为积聚器的基础。更大的挑战在组合器上：其功能是合并两个 Deque 实例，每个都代表对于部分输入的解决方案。根据左侧最后 1 个点到右侧第 1 个点之间的距离，合并有两种可能。如果它们之间的距离非常近，那么左侧最后 1 个段与右侧第 1 个段就必须合并；否则，右侧部分可以附加到左侧。

下面是收集器的 3 个组件提供器、积聚器与组合器的定义：

- 通常情况下，提供器用作容器的构造器；这里可以写成 ArrayDeque::new。不过，正确的初始值是个空容器，会被积聚器或组合器所用。在该示例中，它是一个空的段：

```
Supplier<Deque<Deque<Point>>> supplier =
    () -> {
        Deque<Deque<Point>> ddp = new ArrayDeque<>();
        ddp.add(new ArrayDeque<>());
        return ddp;
    }
```

- 积聚器的作用与在串行代码中是一样的，用于将一个 Point 添加到部分解决方案中：

```
BiConsumer<Deque<Deque<Point>>,Point> accumulator =
    (ddp, p) -> {
        Deque<Point> last = ddp.getLast();
        if (! last.isEmpty()
                && last.getLast().distance(p) > MAX_DISTANCE ) {
            Deque<Point> dp = new ArrayDeque<>();
            dp.add(p);
            ddp.add(dp);
        } else {
            last.add(p);
        }
    }
```

- 合并器会将两个部分解决方案合并到一起：

```
BinaryOperator<Deque<Deque<Point>>,
    Deque<Deque<Point>>> combiner =
    (left, right) -> {
        Deque<Point> leftLast = left.getLast();
        if (leftLast.isEmpty()) return right;
        Deque<Point> rightFirst = right.getFirst();
```

```
        if (rightFirst.isEmpty()) return left;
        Point p = rightFirst().getFirst();
        if (leftLast.getLast().distance(p) <= MAX_DISTANCE ) {
            leftLast.addAll(rightFirst);
            right.removeFirst();
        }
        left.addAll(right);
        return left;
    }
```

图 4-10 展示了提供器、积聚器与组合器的协同工作，用于执行一个示例输入。

可以通过工厂方法 Collector.of 将这 3 个函数放到一个 Collector 中。Collector.of 的 Characteristics 参数用于提供与收集器相关的性能方面的元数据(参见 6.8 节)：

Collector<T,A,R>	Ⓢ
of(Supplier<A>, BiConsumer<A,T>, BinaryOperator<A>, Characteristics...)	Collector<T,A,R>

现在，实现图 4-10 只需要调用 Collector.of 并将生成的收集器提供给 Stream.collect 即可：

```
    Deque<Deque<Point>> displacementRecords =
sortedPointList.stream()
        .collect(Collector.of(supplier, accumulator, combiner));
```

性能

该收集器的并行化做得很好。6.8.2 节会对其性能特性进行深入分析，不过现在可以总结一下：串行执行的性能与之前介绍的迭代版本很接近。如果没有中间操作的成本，那么在一台 4 核机器上，终止流的速度会达到 1.8 倍左右；如果中间操作的成本很高，那么相关的并行加速也会得以改进。

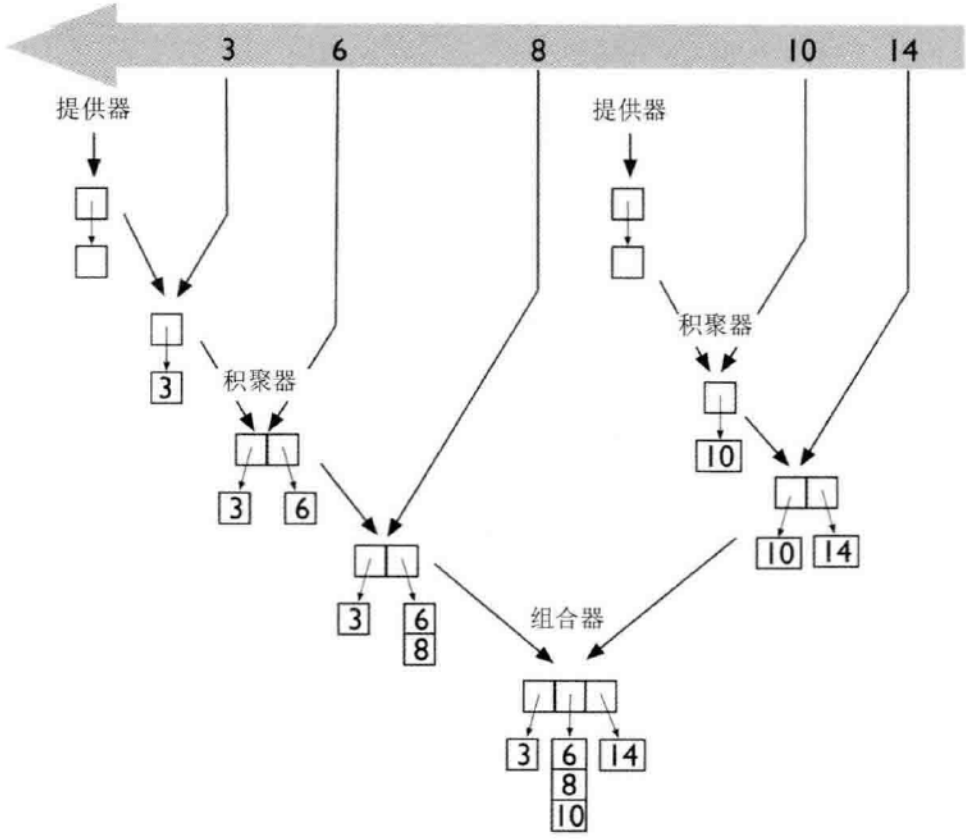


图 4-10　根据临近度对 Point 实例进行收集

4.3.1　完成器

之前讨论的收集器组合需要传递方法 collectingAndThen，它接受一个函数，会在收集完成后应用到容器上：

Collectors	ⓢ
collectingAndThen(Collector<T,A,R>, Function<R,RR>)	Collector<T,A,RR>

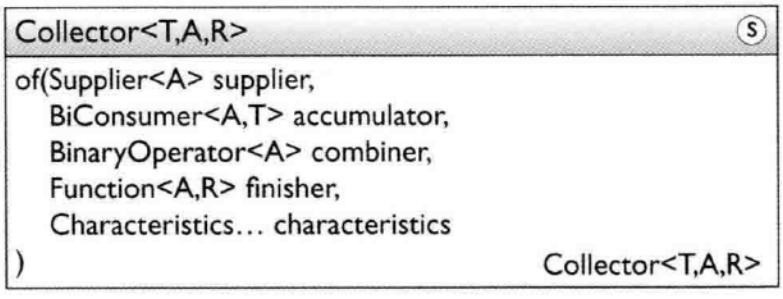
通过该方法，可以为收集器生成的容器提供一个额外的完成转换。例如，要想得到图书标题的一个不变列表，我们可以改变 toList 返回的收集器，如下代码所示：

```
List<String> titles = library.stream()
    .map(Book::getTitle)
    .collect(collectingAndThen(toList(),
          Collections::unmodifiableList));
```

之所以需要完成函数，有下面几个原因：

- 中间容器所返回的类型不正确(例如，用于内部 joining 的 StringBuilder)。
- 有些处理需要推迟到所有元素都可用时才行，例如计算平均数。
- 结果要以规范形式返回，例如重新平衡一棵树。
- 结果在返回前需要以某种形式“封装”起来，例如使用不可修改的或是同步的包装器。

有些收集器的操作总是需要完成器。例如 joining，虽然其结果类型是 String，但它也会被某些收集器所实现，作为不变类型，它是不适合收集的(稍后将会看到一种可用的汇聚形式，不过对于每个积聚或是组合操作来说，其拼接字符串的性能是非常差的)。事实上，用于 joining 的容器是可变类型 StringBuilder；转换为 String 只会发生在完成函数中，在汇聚完毕之后。由于总是需要这个完成器，因此它应该被构建到收集器中；有一个重载的 Collector.of 就是针对这个目的：



我们需要搞清楚 Collector 的第 2 个类型参数的含义：它是积

聚操作所发生的中间容器的类型。在使用收集器时，我们无须关心这个类型，它只与收集器的内部机制有关；通常在编写时，我们只会使用输出类型，就像上一节的示例那样。在下一节的示例中(4.3.2 节)，我们有两个选择，一是使用完成函数将合并器的输出转换为所需类型，二是后续应用一个 map 操作。

为了对一个内建完成器进行可视化，请考虑如下代码，它会创建一系列字符串，每个字符串都包含一本书的所有作者名。

```
Stream<String> concatenatedAuthors = library.stream()
    .map(b -> b.getAuthors().stream().collect(joining());
```

图 4-11 展示了执行的各个阶段。在(a)中，输入字符串中的字符会被积聚到内部构建的 StringBuilder 对象中；在(b)中，输入流被获取完毕，完成器函数会被应用到中间对象上，并且返回结果 String。

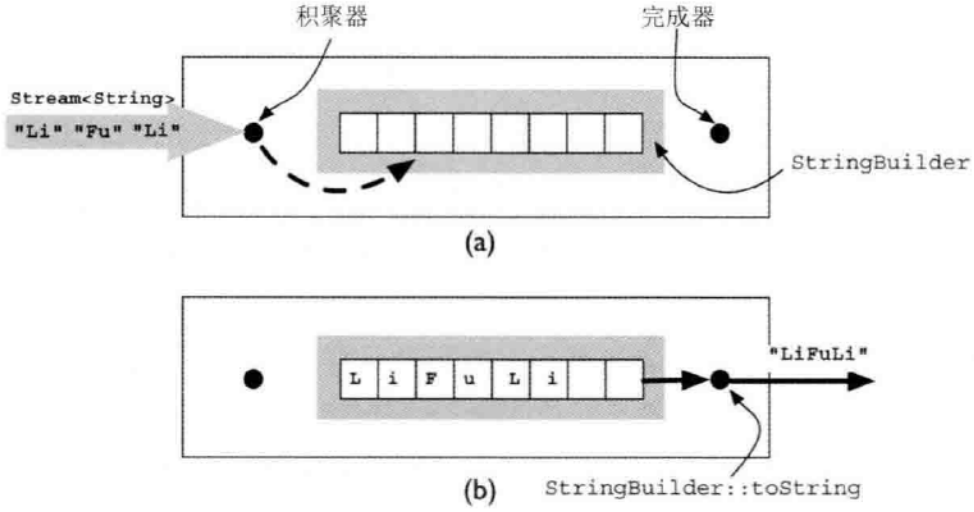


图 4-11　Collectors.joining 的工作原理

将完成函数嵌入到收集器中使得它们非常适合于作为 groupingBy 的下游收集器，而很多时候单独使用完成器则不行。

例如，考虑如下代码，它会创建一个从每个字母到以该字母开头的所有图书标题拼接而成的字符串的映射：

```
Map<Character, String> collect1 = library.stream()
    .map(Book::getTitle)
    .collect(groupingBy(t -> t.charAt(0), joining(";")));
```

如果 joining 的完成器不是收集器的一部分，但却要在 groupingBy 收集之后单独使用，那就有必要对结果映射进行迭代，将完成器应用到每个值上。相反，joining 完成器可以被组合到 groupingBy 完成器中，应用到分类映射的每个值上。

4.3.2 示例说明：找到我的书

本节会通过一个稍微复杂点的示例继续介绍收集器，我们会看看如何计算出书架上每本书的位置，前提是假设它们按照标题的字母表顺序排列。我知道每本书有多少页，每页有多厚，因此如果可以创建一个从每本书的标题到书架上它前面图书的总页数的映射，那么就可以轻松计算出每本书距离书架起始位置的距离。这是状态共享问题的另一个示例，不过更加难以解决(计算代价也更大)：Map中的每个值，除了第1个外，都取决于其前面的值。

解决这个问题的一个主要困难在于要打开思路，思考不同的解决方案：每个 Java 开发者都编写过初始化属性(如累加位移)的程序，然后通过迭代诸如每本书的页数这样的数据来累加变化。这种模式大家都很熟悉了，立刻就会想到解决这类问题的算法一定是串行的。这里我们将会探索另一种利用汇聚的方案来打破这种假设：通过这种方式，我们可以编写出更好的程序，将精力集中在问题的本质上，从而编写出执行速度更快的程序。

这种递归算法的关键步骤在于要设计一种数据结构，它能够持有部分结果，从而组合器的后续应用最终可以将这些结果合并

到最终输出中。不过，组合器还需要对应于部分结果的输出，因此，其数据结构必须要能表示出这两点。

先停下来，仔细思考一下该数据结构的设计。

一本书对应的部分结果就是标题——位移对，与之相关的部分输入则是图书的页数。因此，我们可以得到如下辅助类的定义：

```
class DispRecord {
    final String title;
    final int disp, length;
    DispRecord(String t, int d, int l) {
        this.title = t; this.disp = d; this.length = l;
    }
    int totalDisp() { return disp + length; }
}
```

该示例将会说明为何大家对将元组引入到Java语言中抱有那么大的热情。值类声明很冗长，相比于其他语言中作为元组的值对象来说也缺乏效率。也就是说，我们至少还能做一些工作：定义诸如totalDisp这样的便捷方法可以看作是对必须使用值类的一个补偿。

问题指定了从标题到位移的一个无序 Map，不过每个DispRecord的计算要取决于其前驱，因此在收集过程中需要一个有序容器。我们使用Deque来完成这个目的，这是为了充分利用其访问最后一个元素的便捷性。

现在准备开始编写收集器的组件：

- 提供器的定义很简单：在该示例中，我们只需要通过ArrayDeque::new创建一个空容器即可：

```
Supplier<Deque<DispRecord>> supplier = ArrayDeque::new;
```

- 积聚器有点难度：其任务是将一个 DispRecord 添加到现有 Deque 的末尾。先停下来分析一下其代码骨架。

积聚器会将一个 Book 添加到一个 Deque<DispRecord>中，通过向总页数中添加位移计算出它与 Deque 最后一个元素的位移。

```
BiConsumer<Deque<DispRecord>,Book> accumulator =
    (dqLeft, b) -> {
        int disp = dqLeft.isEmpty() ? 0 :
            dqLeft.getLast().totalDisp();
        dqLeft.add(new DispRecord(b.getTitle(),
            disp,
            Arrays.stream(b.getPageCounts()).sum()));
    };
```

- 现在来考虑合并器。其任务是合并两个Deque<DispRecord>实例。如果还没有想清楚，那么先停下来，写出其代码骨架。

合并器需要在第 2 个元素的位移字段中加上前 1 个元素的总页数。这可以通过最后一个计算出来，方式还是将其位移添加到总页数中。接下来可以合并两个收集了：

```
BinaryOperator<Deque<DispRecord>> combiner =
    (left, right) -> {
        if (left.isEmpty()) return right;
        int newDisp = left.getLast().totalDisp();
        List<DispRecord> displacedRecords = right.stream()
            .map(dr -> new DispRecord(
                    dr.title, dr.disp + newDisp, dr.length))
            .collect(toList());
        left.addAll(displacedRecords);
        return left;
    };
```

现在，问题的主要目标已经达成(可以计算出图书位移了)，不过结果的形式与问题规范所要求的还不一致，我们需要一个从图书标题到位移的映射。实现该需求的一种方式是将我们方才定义的收集器的结果以流的形式放进 Collectors.toMap 所创建的收

集器中。可以向已有的收集器添加一个完成器。先停下来，思考一下该如何定义一个适合的完成器，以及还需要做哪些改变。

在该示例中，完成器的任务只不过是通过其输入创建一个 Map。不过在调用完成器时，fork/join 线程已经完成了合并器的工作，因此它可以用于并发 Map 的合并：

```
Function<Deque<DispRecord>,Map<String,Integer>> finisher =
    ddr -> ddr.parallelStream().collect(
        toConcurrentMap(dr -> dr.title, dr -> dr.disp));
```

图 4-12 展示了我们定义好的 4 个函数，它们用于处理第 3 章的 3 本示例图书。

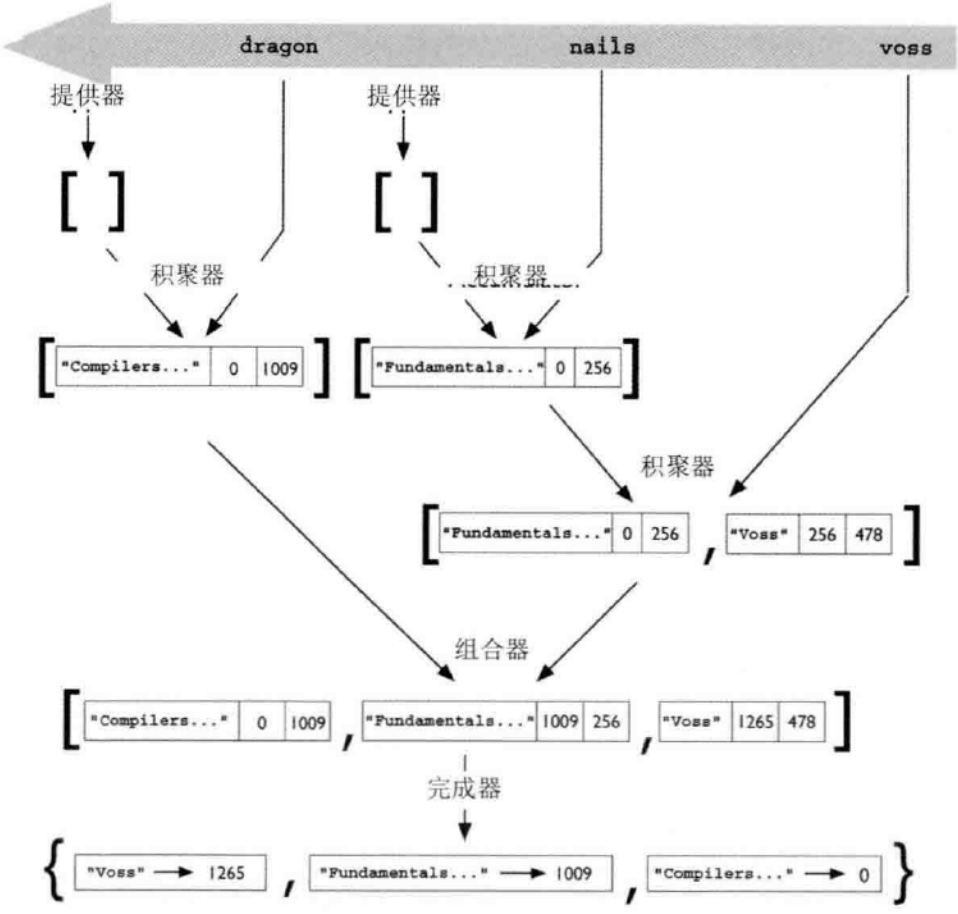


图 4-12　收集书位移

实现图 4-12 的功能只需要调用 Collector.of 并将结果收集器提供给 Stream.collect：

```
Map<String,Integer> displacementMap = library.stream()
    .collect(Collector.of(supplier, accumulator, combiner,
finisher));
```

性能

影响该程序性能的不同因素使得该示例变得很有趣；6.8.3节将会对其进行详细分析。现在简单总结一下：该程序要比迭代版本慢一些，有如下几个原因：这是一个前缀和的示例，其中每个元素值都取决于前面元素的值。在针对前缀和的并行算法中(例如这个)，合并操作的总代价与输入数据集的大小正相关，无论所用的并行级别是什么均如此[1]。这里所用的合并操作代价相当高昂(事实上没必要)。第2个问题是由完成器所执行的Map合并代价造成的；这可以通过预先分配Map来缓和。第3个问题是由程序每个元素过低的负载造成的——在实际情况下，收集前的预处理是很必要的。6.8.3节将会介绍这些问题是否导致该程序不适合于使用并行化。

4.3.3 收集器的规则

当收集器以图 4-12 所示的方式执行时，框架与收集器提供的组件之间就会发生复杂的相互影响。要想使得双方能够正常工作，每一方都要依赖于另一方遵循某些规则。本节将会介绍这些规则：收集器期望框架做什么，以及收集器本身要遵循哪些规则。

框架可以确保遵循以下条件：

---

[1] Java 8提供了java.util.Arrays，它拥有一个新方法parallelPrefix，并且具有不同的重载，用于高效计算前缀和，不过它并没有被添加到Stream API中。

- 新值只能作为积聚器的第 2 个参数；其他所有值都是之前从提供器、积聚器或组合器所返回的结果。
- 来自于提供器、积聚器与组合器的结果可以返回给 collect 的调用者；否则，它们将只能用作积聚器、组合器或完成器的参数。
- 传递给组合器或完成器的没有返回的值永远不会再使用；它们的内容已经得到了处理，不应该再被使用。

收集器必须要遵循如下约束：

- 除非具有特性 CONCURRENT(参见 4.2.1 节)，否则收集器必须要确保从提供器、积聚器或组合器函数所返回的任何结果都是特定于某个线程的(也就是说，其他线程不可访问它)。这样，收集器框架就可以进行并行处理而无须考虑外部线程的干扰了。
- 如果具有特性 CONCURRENT，那么积聚器必须是线程安全的，使用相同的并发可修改的结果容器，因为框架可能会从多个线程中并发调用它。如果需要排序，那么就不能使用并发收集器。
- 同一性约束：在合并元素时，空的结果容器应该保证不会改变其他元素。正式一点说，对于任何值 s：

```
s == combiner.apply(s, supplier.get())
s == combiner.apply(supplier.get(), s)
```

- 结合性约束：在不同位置分割计算应该会得到相同的结果。正式一点说，对于任何值 q、r 与 s：

```
combiner.apply(combiner.apply(q, r), s)) ==
                    combiner.apply(q, combiner.apply(r, s))
```

- 兼容性约束：以不同方式在积聚器与组合器间进行分割计算应该会得到相同的结果。正式一点说，对于任何值 r、s 与 t，左右两侧代码执行应该会得到相同的 r 值：

```
accumulator.accept(s, t);   r = combiner.apply(r, s);
r = combiner.apply(r, s);   accumulator.accept(r, t);
```

4.4 汇聚

在本章一开始，我们比较了收集与其特例汇聚，并且得出一个结论，即在 Java 程序中，收集的用处更大一些。不过，汇聚在某些情况下也是非常有用的；本节将会探索 Stream API 所实现的汇聚的一些用法。

4.4.1 对原生值的汇聚

我们之前已经介绍过了针对原生流特定目的汇聚所提供的便捷方法，包括 sum、min、max、count 与 average。本节将会介绍它们与 reduce 方法一般功能的关系，以及如何通过它定义针对原生流的新功能。本节的示例使用了 IntStream，不过其他原生流类也提供了类似的方法。

针对原生值的汇聚的基本想法遵循了相同的分治法，正如图 4-8 中收集器的 3 个组件所示。使用该图所表示的形式，对原生值做一个简单的汇聚(计算 int 值的和)，如图 4-13，代码如下所示：

```
int sumResult = IntStream.of(1,2,3)
.sum();
```

该图与收集器图存在两个显著差别：

- 一个基值(汇聚的身份)用于替代之前图中提供器函数所创建的空容器实例。
- 积聚器与合并器是相同的，因为只包含一种类型。

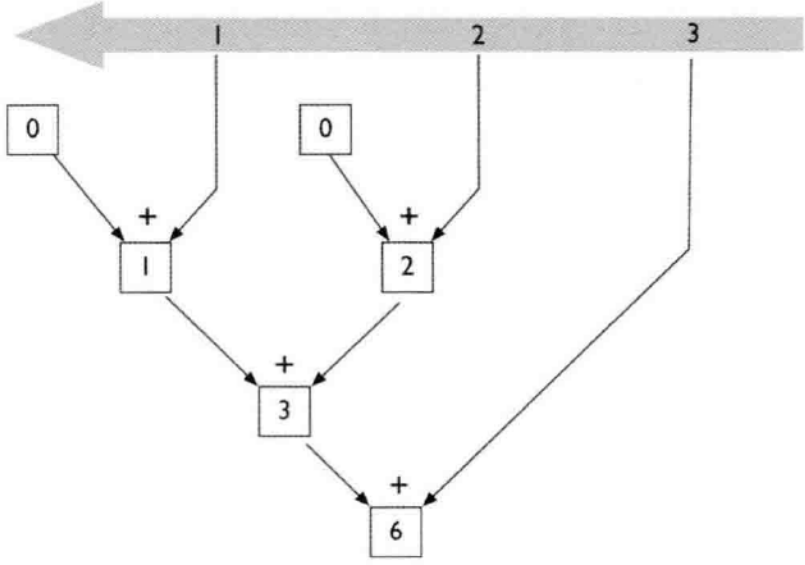


图 4-13　原生值汇聚

这有助于说明 IntStream.reduce 的第 2 个重载方法的签名：

IntStream	ⓘ
reduce(IntBinaryOperator)	OptionalInt
reduce(int, IntBinaryOperator)	int

如果便捷方法 sum 不存在，那么可以通过 reduce 的第 2 个重载方法编写代码来实现图 4-13：

```
int sum = IntStream.of(1,2,3)
.reduce(0, (a, b) -> a + b);
```

在 3.2.4 节介绍的便捷方法中，sum、count 与 average 都以这种方式由 reduce 而来。新函数也可以通过这种方式定义：例如，如下代码会计算变量 intArg 的阶乘：

```
int intArgFactorial = IntStream.rangeClosed(1, intArg)
    .reduce(1, (a, b) -> a * b);
```

给定一个空流，reduce 的这个变种会返回提供的身份。相反，reduce 的第 1 个重载并不会接受一个身份，因此给定一个空流，它必须要返回一个空的 OptionalInt。该变种用于定义便捷方法 max 与 min。如果它们不是 API 的一部分，那么我们可以通过单参数的 IntStream.reduce 达到相同的效果。例如：

```
OptionalInt min = IntStream.of(1,2,3)
    .reduce((a, b) -> Math.min(a,b));
```

该代码的一个执行过程如图 4-14 所示。

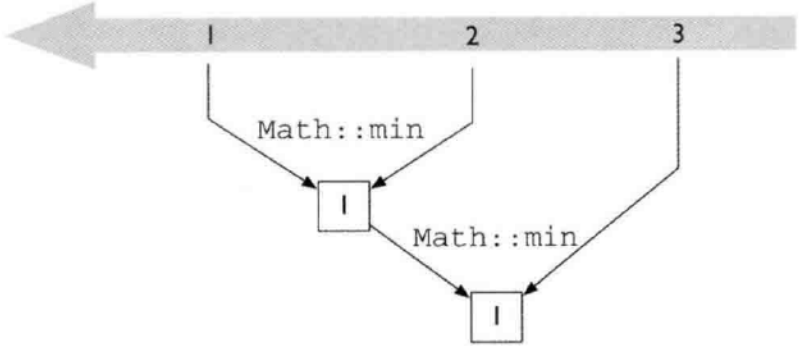


图 4-14　没有身份的原生值汇聚

4.4.2　对引用流的汇聚

介绍完对原生值的收集与汇聚后，我们再来看看对引用流的汇聚，二者之间是非常相似的：Stream.reduce 的两个重载方法类似于原生流的 reduce，大家可能会觉得很奇怪，第 3 个非常类似于收集，不过其使用方式是完全不同的：

Stream<T>	ⓘ
reduce(BinaryOperator<T>)	Optional<T>
reduce(T, BinaryOperator<T>)	T
reduce(U, BiFunction<U,T,U>, BinaryOperator<U>)	U

现在按顺序来介绍每一个重载方法。第 1 个只接受一个合并器，但没有提供身份，这类似于单参数的原生 reduce，它返回一个 Optional。它在某些场景下非常有用：二元运算符(如 Comparator.compare)会返回其中一个操作数。例如，我们可以通过如下代码找到以字母表顺序排列的第 1 本书的标题：

```
    Comparator<Book> titleComparator =
Comparator.comparing(Book::getTitle);
    Optional<Book> first = library.stream()
       .reduce(BinaryOperator.minBy(titleComparator));
```

事实上，这正是便捷汇聚方法 Stream.min 的实现方式。代码的执行可以通过与相应的对原生值的汇聚一样的可视化方式呈现出来(如图 4-14 所示)。二元运算符还可以返回一个新创建的对象，例如调用 BigInteger 或 BigDecimal 的二元数学运算符的结果(注意，我们选择这些类型作为汇聚的示例是因为它们是不可变的)：

```
    Stream<BigInteger> biStream = LongStream.of(1,2,3)
       .mapToObj(BigInteger::valueOf);
    Optional<BigInteger> bigIntegerSum = biStream
       .reduce(BigInteger::add);
```

reduce的所有3个重载方法中的合并器都要遵循与收集器的合并器相同的结合性约束，原因是一样的：对计算在不同位置执行不同的分割必须要返回相同的结果。对于q、r与s的任何值来说：

```
    combiner.apply(combiner.apply(q, r), s)) ==
            combiner.apply(q, combiner.apply(r, s))
```

Stream.reduce 的另外两个重载方法都需要一个身份。它所使用的对象不能被积聚器或合并器修改，因为 reduce 可能会重用相同的对象——这与收集相反，其中 Supplier 在每次调用时都会创建一个新对象。

第 1 个重载接受一个身份和一个二元运算符。我们可以使用它代替单参数的重载方法来计算出 BigInteger 流的总和，而无须返回一个 Optional：

```
BigInteger bigIntegerSum = biStream
    .reduce(BigInteger.ZERO, BigInteger::add);
```

注意这与对原生值汇聚的相似性：我们需要为其提供一个不变身份，如 BigInteger.ZERO。代码的执行可以通过与对应的汇聚原生值的相同方式可视化地呈现出来(如图 4-13 所示)。

与收集器一样，合并器也要遵循同一性约束：给定任何值 s 与身份 id：

```
s == combiner.apply(s, id) == combiner.apply(id, s)
```

第 3 个重载方法引入了积聚器，这样就可以返回不同类型了，例如对某种类型的聚合。如下代码使用它计算出图书馆中卷宗的总数(如图 4-15 所示)。

```
int totalVolumes = library.stream()
    .reduce(0,
            (sum, book) -> sum + book.getPageCounts().length,
            Integer::sum);
```

注意，可以将上述代码写成(通常情况下也是这么做的)单独的 map 与 reduce 操作：

```
int totalVolumes = library.stream()
    .mapToInt(b -> b.getPageCounts().length)
    .sum();
```

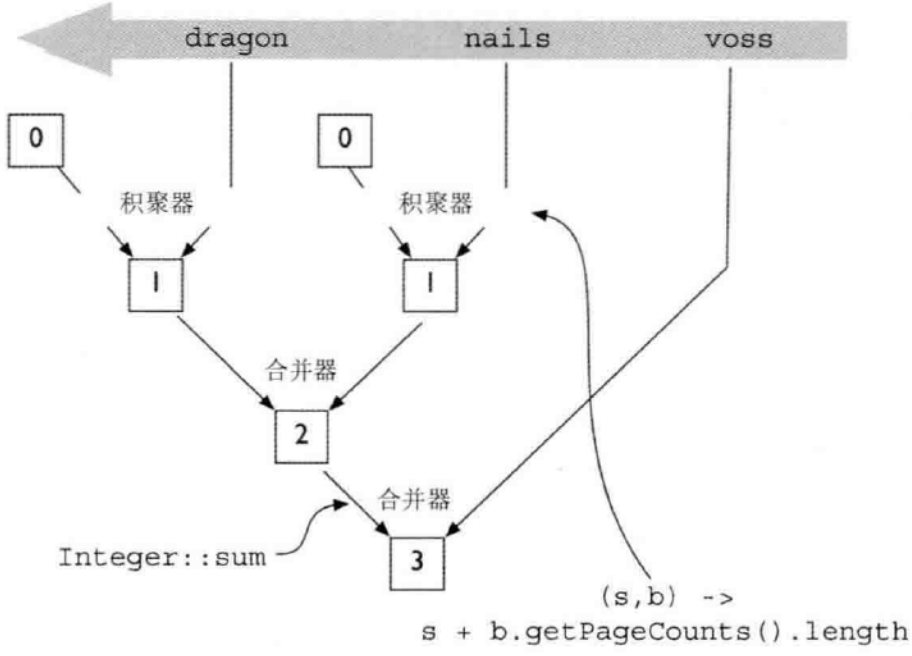


图 4-15　长格式的汇聚

不过，reduce 的这个重载是针对这样的场景的：可以通过将映射与汇聚操作合并到单个函数中来优化性能。

重申一次，与收集器一样，积聚器与合并器的功能必须是兼容的，从而确保一个程序所有可能的执行路径都会得到相同的结果，无论计算是怎么划分的都是如此。不过，这里的情况要简单一些(没有突变的情况永远要简单一些)：对于 r、s 与 t 的任何值来说，如下等式一定是成立的：

```
combiner(r, accumulator(s, t)) == accumulator(combiner(r, s), t)
```

总结一下本节内容，我们应该重新思考一下本章一开始提出的声明，即在 Java 编程中，对于流来说，收集的用处要比汇聚大一些。对身份的重载无法积聚到可变类型中，这一事实毫无疑问使得它们的价值要低于收集。不过，reduce 的单参数重载方法可以达到与收集一样的目的，前提是它要与一个预先的映射阶段组

合使用。出于比较的目的，本节最后对累加页数问题给出了一个汇聚解决方案(已经在 4.3 节通过收集得到了解决)。图 4-16 以可视化形式展现了该程序；将其与图 4-12 进行比较，看看这两种解决方案之间存在哪些差别。

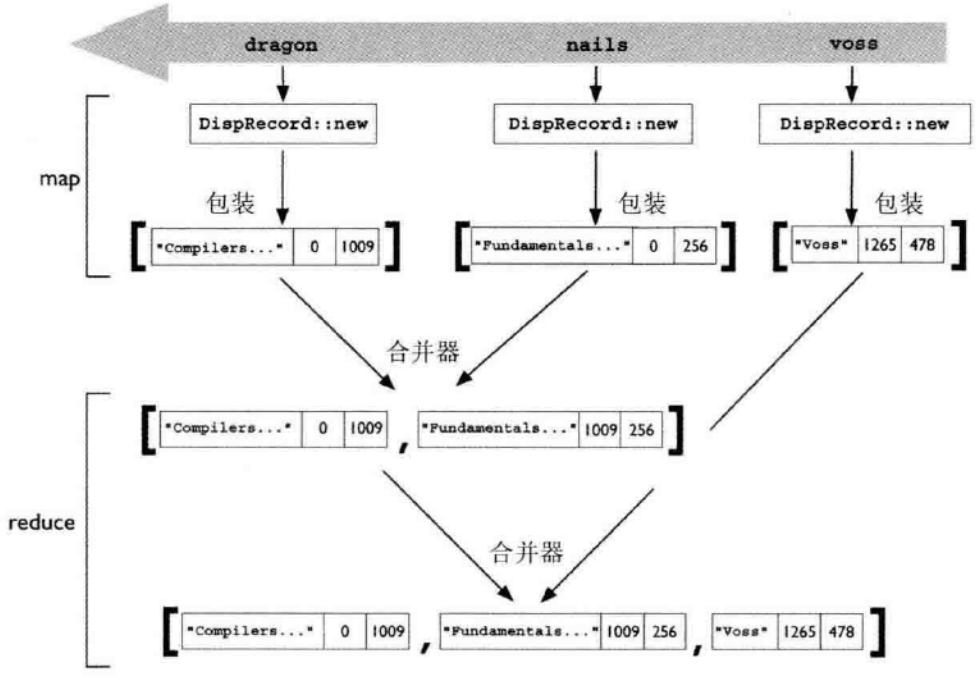


图 4-16　累加的图书位移(map-reduce 版本)

图 4-16 分为两部分，分别被标记为“map”与“reduce”。map 部分完成了收集器版本中提供器与积聚器组件的工作；首先，它会使用一个新的构造器为每个 Book 创建一个 DispRecord：

```
DispRecord(Book b) {
   this(b.getTitle(), 0, IntStream.of(b.getPageCounts()).sum());
}
```

接下来，通过辅助方法wrap将其包装到一个单元素的Deque中：

```
Deque<DispRecord> wrap(DispRecord dr) {
   Deque<DispRecord> ddr = new ArrayDeque<>();
```

```
        ddr.add(dr);
        return ddr;
    }
```

reduce 部分与收集版本中的合并器相同。

最后，客户端代码还有不少工作要做：在该版本中，它必须要组合 map 与 reduce 阶段，并且还要显式处理空流的情况，而非像之前那样将一切都委托给收集器来做：

```
Map<String, Integer> displacementMap = library.stream()
    .map(DispRecord::new)
    .map(this::wrap)
    .reduce(combiner).orElseGet(ArrayDeque::new)
    .stream()
    .collect(toMap(dr -> dr.title, dr -> dr.disp));
```

4.4.3　通过汇聚来组合收集器

4.1.2 节介绍了很多“下游”收集器的示例，其功能与各种终止操作相同，用于与其他收集器组合。其中，我们介绍了 reducing 方法的 3 个重载，它们会返回与 Stream.reduce 的 3 个重载对应的收集器。既然已经理解了 Stream.reduce，那现在回来再简单看看它们：

Collectors	Ⓢ
reducing(BinaryOperator<T>)	Collector<T,?,Optional<T>>
reducing(T, BinaryOperator<T>)	Collector<T,?,T>
reducing(U, Function<T,U>, BinaryOperator<U>)	Collector<T,?,U>

这些收集器的使用与相应的 reduce 重载版本相同，不过它们是用在下游上下文中的，通常作为 groupingBy 收集器的下游。例如，要想找到每个主题中高度最高的书，我们可以使用第 1 个重载版本：

```
    Comparator<Book> htComparator =
Comparator.comparing(Book::getHeight);
    Map<Topic,Optional<Book>> maxHeightByTopic = library.stream()
        .collect(groupingBy(Book::getTopic,
                reducing(BinaryOperator.maxBy(htComparator))));
```

三参数的重载版本的使用方式与此类似。不过要注意，相对于积聚器，该重载版本需要一个 Function<T, U>，其意图类似于上一节介绍的图书位移示例中的 mapping 阶段。例如，可以通过该重载计算出图书馆中每一主题下卷宗的数量：

```
Map<Topic,Integer> volumesByTopic = library.stream()
    .collect(groupingBy(Book::getTopic,
        reducing(0, b -> b.getPageCounts().length,
        Integer::sum)));
```

工厂方法 Collector.counting 就是通过该重载实现的。例如，如果没有提供 Collector.counting，那么可以通过如下代码计算出图书馆中每个主题下图书的数量：

```
Map<Topic,Long> booksByTopic = library.stream()
    .collect(groupingBy(Book::getTopic,
            reducing(0L, e -> 1L, Long::sum)));
```

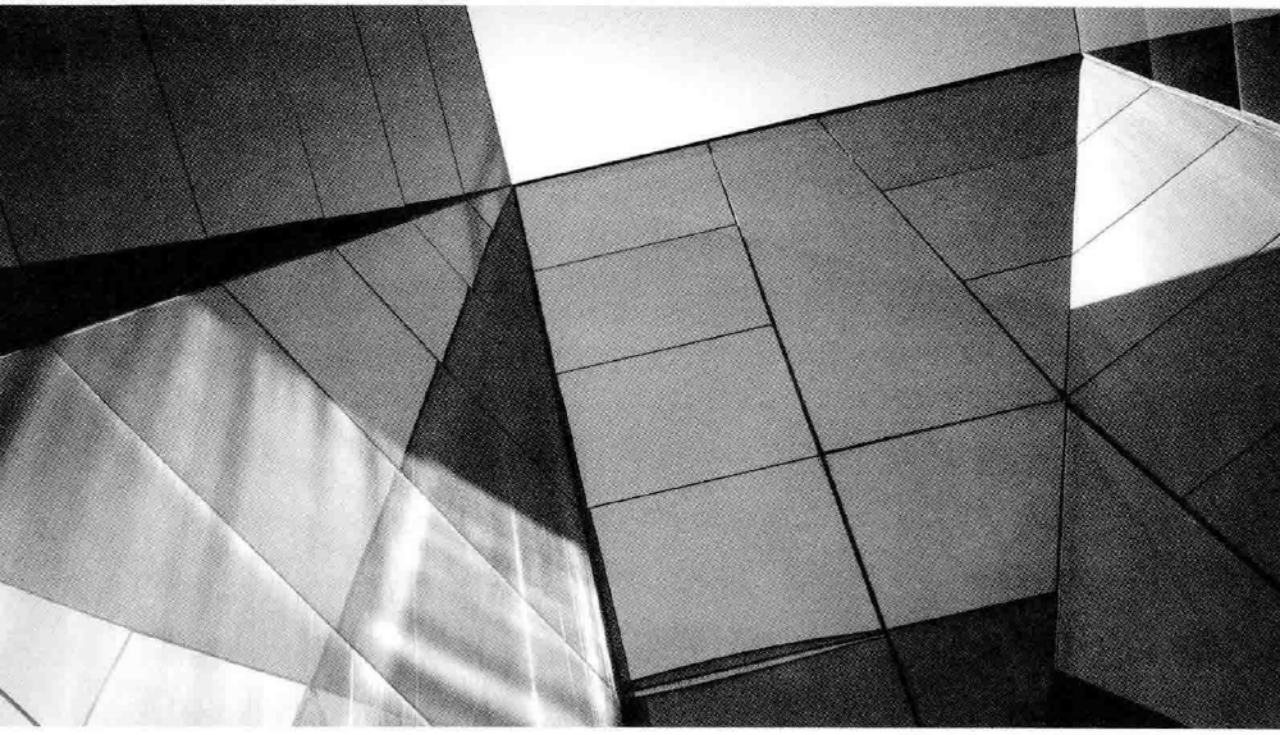
4.5 小结

本章介绍了如何通过 Stream API 获取流处理的汇总结果。总体来说可以分为两类：汇聚与收集。汇聚旨在汇总不变值，而 Stream API 则提供了各种便捷方法来支持汇聚，特别是针对原生值的汇聚。

不过，最重要的技术则是收集，它是汇聚的一般化，可以将并发积聚汇总到可变集合中，甚至还可以管理对那些非线程安全

集合的访问。我们介绍了用于积聚到标准库集合、自定义集合以及分类映射的库收集器。由于收集器本身采用了组合模式，因此库也包含了大量专门用于与其他收集器组合使用的收集器。此外，API 提供了一些扩展点，可以在必要时开发自定义的收集器，我们也介绍了自定义收集器的使用场景以及开发自定义收集器所需的技术。总而言之，收集是一项强大且灵活的工具。掌握收集对于成为 Stream API 专家是非常重要的。

我们对终止流的探索到此为止，下一章将会介绍如何开启流。



第5章

起始流：源与分割迭代器

第 3 章简要介绍了流源这个主题，不过后续的流处理示例要么使用集合，要么使用流工厂方法作为源。这么做反映了主流的做法，不过现在是时候探索关于流创建其他方面的主题了。本章将会介绍如下内容：

- 平台类提供的流方法
- 对流源与中间操作所抛出的异常的处理
- 流源的工作机制
- 涵盖以上主题的一个示例

5.1 创建流

除了Collection接口的流方法外，也许最重要的创建流的方式就是使用流接口本身的工厂方法了。其中，Stream.empty与Stream.of是在第3章介绍的，下一组包含了iterate与generate：

Stream<T>	Ⓢ
iterate(T, UnaryOperator<T>)	Stream<T>
generate(Supplier<T>)	Stream<T>

方法iterate接受一个种子和一个函数(由函数式接口UnaryOperator表示)，然后重复应用该函数来生成连续的流元素。类似的iterate方法被所有的引用流所定义。原生流也有类似的方法。例如，要想创建一个IntStream，并且使其值在 1 与-1 之间交替变换，我们可以通过如下代码实现：

```
IntStream alternatingSigns = IntStream.iterate(1, i -> -i);
```

流中的每个元素都是通过对前一个元素应用该函数生成的。iterate 所生成的流是无限的：只有使用可以被应用到有限初始子流的操作才能够从中获得有用的结果，就像 limit 与短路的“搜索”操作。

方法 generate 接收一个 Supplier，它代表一个无须输入即可生成值的函数，然后重复调用它来生成连续的流元素。生成的串行流是无序的，因此 generate 旨在用于不要求顺序的场景下。例如，它可用于生成常量流或是自定义的随机流。

第 3 组中的工厂方法会创建一个有序流，其中包含一个值范围。它们只定义在整型原生类型 IntStream 与 LongStream 中。下面是其 IntStream 版本：

IntStream	Ⓢ
range(int, int)	IntStream
rangeClosed(int, int)	IntStream

API 中包含了 range 与 rangeClosed，前者不包含指定的结束值，后者则会包含。因此，我们可以通过两种不同的方式来创建相同的流，如下代码所示：

```
IntStream.range(1, 6).forEach(System.out::print);
   //prints 12345
IntStream.rangeClosed(1, 5).forEach(System.out::print);
   //prints 12345
```

这些方法的一个应用场景就是用来模拟索引流。Stream API 并没有对索引流提供直接的支持，不过通过将 IntStream.range 与 IntStream.rangeClosed 的元素映射到索引集合或是数组上，我们可以得到同样的效果。例如，想要图书馆中图书的一个列表，展示出每本书的卷号与页数，就像下面这样：

```
Fundamentals of Chinese Fingernail Image: 1:256
Compilers: Principles, Techniques and Tools: 1:1009
Voss: 1:478
Lord of the Rings: 1:531, 2:416, 3:624
```

这可以通过 IntStream.rangeClosed 实现：

```
library.stream()
   .map(book -> {
      int[] volumes = book.getPageCounts();
      return
         IntStream.rangeClosed(1, volumes.length)
            .mapToObj(i -> i + ":" + volumes[i - 1])
            .collect(joining(", ", book.getTitle()
               + ": ", ""));
   })
```

```
    .forEach(System.out::println);
```

这种将 int 范围的元素作为索引的技术还可以扩展到多个数据源，为 Stream API 中缺乏将两个流“联系”起来的操作提供了解决方案。

最后一个 Stream 方法并非流创建器，而是流合并器，它是一个静态方法 concat，它会将已有的两个流串联起来创建一个新的流：

Stream<T>	Ⓢ
concat(Stream<T>, Stream<T>)	Stream<T>

本节后续部分将会介绍添加到其他平台类中的方法，这些方法会以新方式公开出待处理的数据。

java.util.Arrays

该类提供了针对引用类型的流数组以及所有原生流类型数组的方法。对于每一种流类型来说都有两个重载方法，一个用于流化整个数组，另一个用于流化数组的一部分，其中包含开始索引对应的元素，但不包含结束索引对应的元素。

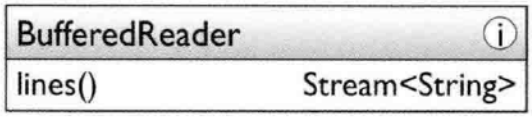
Arrays	Ⓢ
stream(T[ ])	Stream<T>
stream(T[ ], int, int)	Stream<T>
stream(int[ ])	IntStream
stream(int[ ], int, int)	IntStream
stream(long[ ])	LongStream
stream(long[ ], int, int)	LongStream
stream(double[ ])	DoubleStream
stream(double[ ], int, int)	DoubleStream

从表面上看，接受整个数组的方法(而非接受数组的一部分)

在功能上与 Stream.of 的各种特化重叠了，对于 Stream.of 来说，其可变参数可以传递一个数组参数。不过，Stream.of 的真正目的旨在用于需要提供确定且已知数量参数的情况下，如果调用时提供了数组，那么实际上它会被解释为一系列单独的值而非单个数组。因此，对于数组参数来说，请使用 Arrays.stream 来避免这个小问题。

java.io.BufferedReader

该类会读取文本文件；在 Java 8 之前，它主要是通过方法 readLine 发挥作用的，每次调用 readLine 都会返回一行。在 Java 8 中，它还声明了方法 lines：



该方法会延迟装配流，如果下游操作需要值时，它会调用 BufferedReader.readLine 装配。不要将这个方法与其他任何操作搭配使用，否则当流的终止操作执行时，读取器就无法被访问了，当终止操作完成后，我们对于其状态也无从得知。如果想在调用 lines 后继续使用读取器，那就需要先将其重置为之前设置的标记处。

lines 所调用的 BufferedReader 方法在各种错误情况下会抛出 IOException 异常，例如在读取器已经关闭的情况下调用它们。相对于强制调用者处理这些检查的异常，lines 将其包装到了一个 UncheckedIOException 异常中，触发读取的终止操作会抛出这个异常。下一节将会探索这么做的效果。

第 6 章将会看到，在影响并行流性能的诸多因素中，源可达到的并发级别占据了很重要的地位。从 BufferedReader 这样的流

源所发出的数据本质上是串行的，因此库所采取的分割策略是在可能的情况下将其划分为批处理形式。如果以这种方式读取数据是处理过程中代价最为高昂的部分，那么并行化所能带来的好处就微乎其微了。不过，如果分割迭代器在分割前能够缓存数据，那就可以向并行流提供数据了。本章最后将会介绍一个用于代替串行文件读取的有用示例。

java.nio.file.Files

该类提供了对文件与目录的操作，这通常会被委托给本地文件系统。为了保持与类中已有方法的一致性，如果无法打开文件，那么这些方法会抛出 IOException 异常，不过后续所有 IOException 会被包装到 UncheckedIOException，就像 BufferedReader 一样。

第 1 组的 4 个方法会以流的形式生成目录列表：

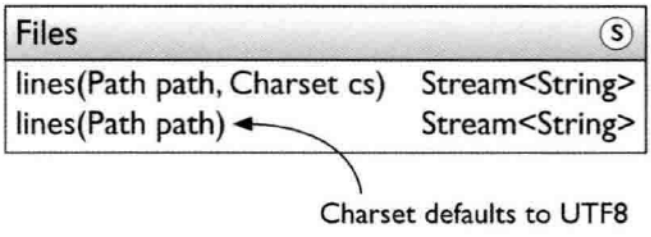
Files	Ⓢ
walk(Path, FileVisitOption...)	Stream<Path>
walk(Path, int, FileVisitOption...)	Stream<Path>
find(Path, int, BiPredicate<Path,BasicFileAttributes>, FileVisitOption...)	Stream<Path>
list(Path)	Stream<Path>

walk 与 find 所生成的列表是递归的，也就是说，它们会包含起始目录的所有子目录；list 所生成的目录只会用作起始目录。这些递归方法会接收 0 个或多个类型为 FileVisitOption 的参数，这是一个枚举，用于对遍历进行配置，例如，指定是否要考虑符号链接等。另外，它们还存在一些差别，例如是否支持最大的遍历深度，是否接受某些谓词，根据 BasicFileAttributes(例如修改时间、访问时间，大小等)来过滤访问路径等。

这些方法所返回的流会间接地封装本地文件句柄，因此需要在 try-with-resources 结构中分配它们。例如，如下代码会递归地访问目录树，从当前目录开始，打印出每个文件的详细信息：

```
Path start = new File(".").toPath();
try(Stream<Path> pathStream = Files.walk(start)) {
    pathStream
        .map(Path::toFile)
        .filter(File::isFile)
        .map(f -> f.getAbsolutePath() + " " + f.length())
        .forEachOrdered(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

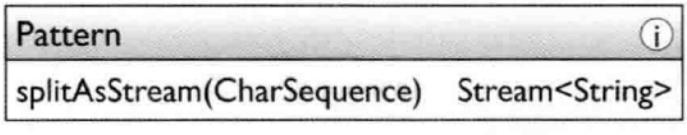
最后两个 Files 方法可以将一个文件解析为文本行；它们是对 BufferedReader 方法的包装，提供了便捷的访问。由于创建一个 BufferedReader 需要一个 Charset 来管理字节到字符的转换，因此还需要为新的 Files.lines 方法提供一个 Charset：



显然，Files.lines 与其底层的 BufferedReader 实现具有相似的性能特性。

java.util.regex.Pattern

该类之前已经有了一个名为 split 的便捷方法，它会返回一个数组；在 Java 8 中，该类又增加了一个类似的流方法 splitAsStream：



类似于 split，该方法也会将输入划分为子字符串，每个子字

符串以匹配该模式的子序列结束，或以输入的结尾作为结束。如下示例展示了不同模式与输入字符串的起始位置相匹配的效果。方法 exampleSplit 返回的字符串展示了对 ls -l 的输出应用了模式后的结果。

```
String exampleSplit(String pattern) {
    Pattern p = Pattern.compile(pattern);
    String s = "-rw-rw-r-- 1 root admin 21508 26 Feb 2014
/.bashrc";
    return p.splitAsStream(s)
        .collect(joining("\",\"", "\"", "\""));
}
```

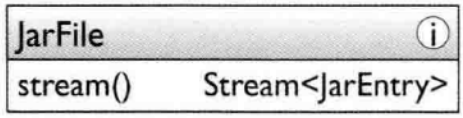
如果模式没有匹配输入的任何子序列，那么结果流就只会包含与输入序列对应的单个元素(参见表 5-1 的第 2 行)。输入序列开头非 0 宽度的匹配会在流的起始位置处插入一个空白字符串(第 3 行)，而 0 宽度的匹配则不会(第 4 行)。

表 5-1 exampleSplit 的行为

输 入	输 出
"o+"	"-rw-rw-r-- 1 r","t admin 21508 26 Feb 2013 /.bashrc"
"@"	"-rw-rw-r-- 1 root admin 21508 26 Feb 2013 /.bashrc"
"^-"	"","rw-rw-r-- 1 root admin 21508 26 Feb 2013 /.bashrc"
"^"	"-rw-rw-r-- 1 root admin 21508 26 Feb 2013 /.bashrc"

java.util.jar.JarFile

到目前为止，jar 文件中的内容只能通过 InputStream 或是 Enumeration 读取；现在则增加了更为便捷的流方法：



java.util.zip.ZipFile

添加到该类中的新方法类似于 JarFile：

ZipFile	ⓘ
stream()	Stream<ZipEntry>

java.lang.CharSequence

该接口的实现有 String、StringBuffer、StringBuilder 与 java.nio.CharBuffer。Java 8 又添加了两个流方法，chars 与 codePoints，其行为与经常使用的Unicode字符完全一致：

CharSequence	ⓘ
chars()	IntStream
codePoints()	IntStream

这两个方法之间的差别在于对 Unicode 补充字符的处理上，即码位大于 U+FFFF 的那些字符。补充字符包括东亚语言的表意文字以及图像字符(包括 emojis)。出于效率的原因，一般来说，相对于 codePoints，我们更倾向于使用 chars，不过与 codePoints 不同，它无法正确解释补充字符。

新方法会返回IntStream(在缺少专门的CharStream的情况下的最好选择)，因此有必要搞清楚如何将 int 值转换为 Stream<Character>。关键在于，因为需要装箱，所以需要告诉 Stream.mapToObj正确的原生类型才行：

```
Stream<Character> chrStream = "example".chars().mapToObj(i ->
(char)i);
```

java.util.BitSet

该类的 toString 方法总是返回被设置位的索引的字符串表示。现在，stream 方法会以一种容易处理的方式返回这些索引。

BitSet	ⓘ
stream()	IntStream

例如：

```
byte[] bits = {10, 18};  // 01010000 01001000 (little-endian)
BitSet bs = BitSet.valueOf(bits);
System.out.println(bs.stream()
    .boxed().collect(toList())); // prints [1, 3, 9, 12]
```

java.util.Random 与 java.util.SplittableRandom

4 个 Random 类 (Random 有两个子类， SecureRandom 与 ThreadLocalRandom)都公开了相同的方法，用于生成随机流。根据所返回的原生流类型，这些方法被分为3组：

Random, SplittableRandom	ⓘ
ints()	IntStream
ints(int, int)	IntStream
ints(long)	IntStream
ints(long, int, int)	IntStream
longs()	LongStream
longs(long, long)	LongStream
longs(long)	LongStream
longs(long, long, long)	LongStream
doubles()	DoubleStream
doubles(double, double)	DoubleStream
doubles(long)	DoubleStream
doubles(long, double, double)	DoubleStream

每组都有 4 个方法，提供了不同的选择组合：返回固定或无限数量的值，每个值是否在定义的范围内。Java 8 引入的类 SplittableRandom 用于支持并行生成随机数。它通常会用在 Spliterator(用于创建流的对象，本章后面将会对其进行详细介绍)

中，给定创建随机数的需求后，它会将这些随机数划分为不同段，并且为每个段中随机数的创建指定单独的生成器。这种划分用到了方法 SplittableRandom.split，它会构建并返回一个新实例，新实例与原实例不会共享任何可变状态，不过会像之前那样使用同样的统计属性来创建一系列值。

虽然上述列表包含了平台库中所有重要的流方法(实际上是最常用的流源)，不过这并不是全部：本章后面将会介绍如何通过定义一个 Spliterator 以从任何数据源来创建自己的流。

5.2　分割迭代器与 Fork/Join

本节将会继续介绍流源，探究其工作方式。你可以在不理解其创建方式的情况下高效使用流，就像之前的示例所展示的那样。不过，你可能想知道流创建的机制会对性能产生何种影响(第 6 章将会对此进行介绍)，如果想从非平台类的源创建自己的流，那么理解流的创建机制就是非常有帮助的了。下一节将会介绍一个简单但却实际的示例。

首先要介绍的是并行分治算法，它是在第 1 章介绍流的时候引入的，同时还会介绍 fork/join 框架是如何实现它的。在并行流时代，框架所提供的主要类是 ForkJoinTask 的一个子类：该类的实例会包装一个流处理任务，这包含了待处理的数据以及要对数据的每一个元素所施加的处理动作。任务执行会采取两种路由之一：数据以遍历的方式在调用线程中被处理，或是先分解，然后从 ForkJoinPool 分配的不同线程中创建新任务来处理部分数据，剩余数据会被分配给现有任务，然后在调用线程中再次执行。

因此，任务所使用的数据访问器就必须支持这两种方式：分

割与迭代直接执行。该数据访问器的类型也是根据这两个函数命名的：java.util.Spliterator。Spliterator 的两个关键方法与其两个关键功能相对应：trySplit(分割自身，在自身与新的 Spliterator 之间划分数据)与 tryAdvance(接受一个 Consumer，并将其应用到下一个元素上)。在深入介绍 Spliterator 之前，我们先来看看它是如何与 fork/join 框架搭配使用的。

如下伪代码对方才介绍的 fork/join 任务做了极大的简化。在该代码中，T 是流要处理的元素类型，方法 makeAbstractTask 会创建一个新任务，设定其 spliterator 字段。值 sizeThreshold 已经被框架计算过了，并作为值得并发执行的最小任务数据量，这需要考虑到总的原始数据量与处理环境：

```
Spliterator<T> newSpliterator;
while (this.spliterator.estimateSize() > sizeThreshold &&
    (newSpliterator = this.spliterator.trySplit()) != null) {

// the f/j framework and the spliterator have both agreed to a
// split, and the spliterator's data has been divided between the
// the new spliterator and itself; now wrap a new task around
// the part of the data assigned to the new spliterator and
// assign it for execution in another thread.

    makeTask(newSpliterator).fork();
}
// process remaining data by iteratively calling
// this.spliterator.tryAdvance() in this thread
```

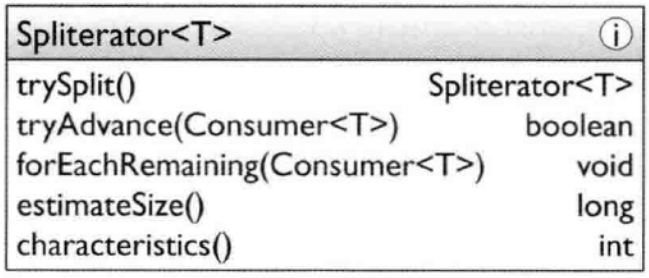
上述代码的循环条件表明要执行分割而不是迭代处理需要满足两个条件：

- 新派生出来的任务可以利用当前未使用的处理能力。在 Java 8 中，sizeThreshold 的值是总数据量除以可用线程数

的结果，不过未来框架可能会使用更复杂的标准来确定其值。

- 需要满足数据结构本身的条件才能使分割有意义。调用 trySplit 时可能会返回 null 而非新的 Spliterator，也许是因为数据集太小，从并行所获得的收益无法抵偿掉分割本身的代价，可能还会有其他原因。

下面是 Spliterator 接口的主要方法：



除了 forEachRemaining 外，其他方法都是抽象的。下面会简要介绍每一个方法。5.4 节将会介绍一个自定义分割迭代器，用于帮助你进一步理解分割迭代器的工作原理，以及何时应该编写自己的分割迭代器。

trySplit

之前曾经介绍过，该方法会创建一个新的 Spliterator，并将自己的一些元素(理想情况下是一半)放到里面。达不到这个理想值的分割器可能也是高效的；例如，分割一个平衡二叉树依然可以实现很好的并行。另一方面，我们有 BufferedReader，它无法实现相等的分割，在极端情况下，有些类根本无法分割，其分割迭代器也总是拒绝分割(这也是方法 Collection.parallelStream 会返回一个“可能是并行”流的原因所在)。

如果顺序对于流源来说很重要，那么执行完 trySplit 之后，分

割迭代器所涵盖的元素就必须接替它所返回的分割迭代器所涵盖的元素。

tryAdvance

该方法将 Iterator 的方法 next 与 hasNext 的功能组合到了一起。如果没有剩余元素，那么 tryAdvance 会返回 false；否则，它在调用 Consumer 的 accept 方法(将下一个元素作为参数)后会返回 true。这是分割迭代器相较于迭代器的两个主要改进之核心：对于每个元素来说所需的方法调用次数更少，性能改进更为明显，同时消除了在调用 hasNext 与 next 方法之间集合发生改变的竞态条件风险。

forEachRemaining

提供了一种分块遍历的机制：相对于处理完每个元素后就返回的做法(就像 tryAdvance 那样)，该方法会在一次调用中处理完分割迭代器中所有余下的元素。这么做会消除每次调用 tryAdvance 的代价，从而提升了性能。其默认实现只是重复调用 tryAdvance，直到返回 false 为止。

estimateSize

返回分割迭代器所涵盖的元素的一个数量估计值。如果分割迭代器提供的是无限流，或是计算元素数量的成本太高，那么 estimateSize 就会返回 Long.MAX_VALUE。不过，即便是不精确的估算值通常也是有用的，而且计算成本也不会很高。

characteristics

返回该分割迭代器的特性集合(参见 6.3 节)。

来自分割迭代器的流

分割迭代器提供了流的本质之所需。要想创建流，可以使用实用工具类 StreamSupport，它提供了接受一个 Spliterator 方法来创建引用流，或是接受 Spliterator 专门的原生子类型(Spliterator.ofInt、Spliterator.ofLong 或 Spliterator.ofDouble)来创建原生流。本章最后一节将会介绍一个实际的示例，它会从一个自定义的分割迭代器来创建流。

不过，值得注意的是，StreamSupport.stream 也可以作为 Collection 接口的流方法的替代者(可能不是特别恰当)。这是因为每个 Collection 实例都是 Iterable 的一个实现，它公开了方法 spliterator。因此，对于任何 Iterable<T>、iter 来说，你都可以这样编写：

```
Stream<T> str = StreamSupport.stream(iter.spliterator(),
false);
```

其中参数 false 表示在该示例中，生成的流是串行的而非并行的。这实际上正是 Collection 的流方法的实现方式。

Iterable.spliterator的默认实现性能很差，这是因为任何好的分割策略都依赖于特定Iterable的物理结构。因此，实现通常会将其重写，正如Java Collections Framework的类所做的那样。不过，如果使用的API接受或是返回Iterable而非Collection实例，那么你就要知道这种直接执行流处理的方式，而不是将其转储到集合中。

5.3　异常

2.7.2 节曾介绍过，lambda 表达式对于管理检查的异常并没有

提供特别的机制；lambda 抛出的任何检查的异常都必须要通过其函数类型进行显式声明。这样，检查的异常就与 Stream API 中实现的延迟计算不太匹配。本节将会探究这个问题；此外，还要说一下为何要将本节内容放到讲解流创建的章节中来：在流处理中，最常使用的检查的异常就是那些将文件用作流源的 API。

为了理顺这些问题，我们来看几个复杂性逐步上升的场景。先来看一个基本的用例，我们想要列出某个目录下每个文本文件的内容。想法是使用 Files.list(提供一个目录路径)生成一个文件路径流，每个对应到目录中的一个文件；然后按顺序将其提供给 Files.lines，后者则会根据每一个来生成一系列文本行。可以通过 flatMap 将这些流拼接为单个流。如果不考虑异常，那么我们可以像下面这样编写代码(其中的参数 start 表示任意目录路径)：

```
Files.list(start)
    .flatMap(Files::lines)
    .forEachOrdered(System.out::println);
```

这是本章后续示例(5.4 节)的基础：grep(一个 Unix 实用工具，将文本文件中的行与正则表达式模式进行匹配)的一个 Java 实现。

由于问题围绕的是异常抛出的点，因此先从基本情况开始，其中异常抛出完全没有被推迟。如下代码创建了一个 Path 对象流，对应于路径为 start 的目录的内容。出于演示的目的，我们只是创建一个流，但不使用它，异常处理会被委托给调用者，这是通过重新抛出包装在 RuntimeException 子类 UncheckedIOException 中的检查的异常实现的。不过，首先打印出栈跟踪信息，看看到底发生了什么：

```
try (Stream<Path> paths = Files.list(start)) { // line 19
} catch (IOException e) {
    e.printStackTrace();
```

```
            throw new UncheckedIOException(e);
        }
```

如果 start 不是一个可以打开并读取的目录路径，那么上述代码会抛出 IOException 异常并被捕获到。例如，如果目录权限不允许访问 start 目录，那么栈跟踪将如下所示：

```
java.nio.file.AccessDeniedException: ./fooDir
    <omitted: frames for platform-specific filesystem access
methods>
    at java.nio.file.Files.newDirectoryStream(Files.java:457)
    at java.nio.file.Files.list(Files.java:3448)
    at ExceptionsExample.main(ExceptionsExample.java:19)
```

这与我们对异常工作方式的理解是一致的，但也许与我们对流的延迟计算的期望不太一致。没有对 Files.list 所创建的流调用终止方法，因此并没有计算其元素。不过尝试打开目录./fooDir 的操作却提早发生了。

接下来，假设目录已经打开了，并且路径流也已经成功创建了。现在，flatMap 会调用 Files.lines，这会先打开每个文件，然后将其内容读取到一系列行中，最后将其追加到返回的流中。如果尝试打开其中一个文件失败了，那会怎么样呢？正如对 Files 的介绍所谈到的那样，诸如 Files.lines 这样的基于 I/O 的流创建方法在这种情况下会抛出检查的异常 IOException。不仅仅是目录，我们还可以让一个文件无法访问来展示该问题；为了保持中间操作的简单性，可以使用 peek 来计算行为型参数：

```
try (Stream<Path> paths = Files.list(start)) {
    paths.peek(path -> {
        try {
            Files.lines(path);          // line 42
        } catch (IOException e) {
            System.err.println("** exception from Files.lines");
```

```
            e.printStackTrace();
        }
    })
    .forEach(line -> {}); // line 47
} catch (IOException e) {
    System.err.println("++ exception from Files.list");
    e.printStackTrace();
    throw new UncheckedIOException(e);
}
```

上述代码会生成如下输出：

```
** exception from Files.lines
java.nio.file.AccessDeniedException: ./fooDir/barFile
    <omitted: frames for filesystem access methods>
    at java.nio.file.Files.lines(Files.java:3782)
    at ExceptionsExample.lambda$main$0
        (ExceptionsExample.java:42)
    at ExceptionsExample$$Lambda$2/1149319664.accept
        (Unknown Source)
    <omitted: frames for stream implementation methods>
    at java.util.stream.ReferencePipeline.forEach(
        ReferencePipeline.java:418)
    at ExceptionsExample.main(ExceptionsExample.java:47)
```

请注意 forEach 的栈帧。这与我们的理解相一致，即终止操作的执行触发了管道元素的计算。

最后，我们可以通过 flatMap 代替 peek 的调用，flatMap 会接收每个文件的字符串流，并将其合并到单个流中：

```
try (Stream<Path> paths = Files.list(start)) {
    paths.flatMap(path -> {
        Stream<String> lines;
        try {
            lines = Files.lines(path);
        } catch (IOException e) {
            e.printStackTrace();
```

```
            lines = Stream.of("Unreadable file: " + path);
        }
        return lines;
    })
    .forEach(line -> {});     // line 57
} catch (IOException e) {
    e.printStackTrace();
    throw new UncheckedIOException(e);
}
```

在这个版本的代码中，打开文件失败会生成与上一个示例非常相像的栈跟踪。不过，假设 Files.lines 成功打开了文件并返回一个流，但没有计算所包含的行。一旦文件打开，终止操作就调用压平代码将独立的行流拼接到单个流中。为了做到这一点，需要对流进行计算并读取已经打开的文件；这个动作也可能遭遇失败。不过，lambda 表达式(其执行会打开文件)现在却超出了作用域，并且文件读取代码也是直接调用的，因此失败通知必须要通过非检查异常来公开。为了强调这一点，我们可以对管道代码进行分解，将其划分为不同阶段，每个阶段都抽取出一个局部变量：

```
Stream<Path> paths = Files.list(start);
Stream<String> lines = paths.flatMap(path -> ...);
lines.forEach(line -> {});
```

由于当lambda表达式执行失败时，flatMap不在调用栈中，因此其抛出的异常信息与实际情况并没有什么关系。出于这个原因，用于创建流的平台库代码总会将检查的异常包装到非检查异常中。最常见的就是将IOException包装到新的java.io.UncheckedIOException中；当文件打开后发生了失败情况，它就会被抛出来。对于这种失败来说，其栈跟踪如下所示：

```
Exception in thread "main" java.io.UncheckedIOException:
    java.nio.charset.MalformedInputException: Input length = 1
```

```
    at java.io.BufferedReader$1.hasNext(BufferedReader.java:574)
    <omitted: frames corresponding to stream implementation methods>
    at java.util.stream.ReferencePipeline.forEach(
    ReferencePipeline.java:418)

    at ExceptionsExample.main(ExceptionsExample.java:57)
Caused by: java.nio.charset.MalformedInputException:
    Input length = 1
at java.nio.charset.CoderResult.throwException
    (CoderResult.java:281)
    <omitted: frames for text file reading and charset decoding>
    at java.io.BufferedReader$1.hasNext(BufferedReader.java:571)
    ... 18 more
```

该栈信息是由某一个行流在计算时抛出了MalformedInputException异常所引起，这是在通过Charset处理文件，但遇到了无法解码的Unicode字符时抛出的(例如，本来要读取的是二进制文件，但实际读取的却是文本文件)。该检查的异常会被BufferedReader的hasNext方法所捕获(从Files.lines调用)，并且会被包装到UncheckedIOException中，这个异常在调用栈顶部是没有被捕获的。检查的异常堆栈底部的“18 more”栈帧内容与上面是相同的，并且一同构成了整个非检查的异常栈；出于简化的目的，重复的内容被略掉了。

因此，管道操作内部所调用的延迟计算代码只会抛出非检查的异常，这会结束终止操作，并停止管道处理。如果想要从错误中恢复，那么必须要通过检查的异常来通知，这些检查的异常来自于立即计算的操作。例如，之前曾看到过，尝试从二进制文件读取字符会导致异常MalformedInputException；对于当前的用例来说(打印目录中每个文本文件的内容)，存在一个简单的恢复动作：忽略该文件。不过，只有在抛出该异常的方法是立即计算时才能这么做，就像Files.readAllLines一样，它会立即读取整个文

件，如果遇到了无法解码的字节序列时则会抛出一个检查的异常。这种做法构成了下一节将会介绍的递归grep问题解决方案的基础。

5.4　示例说明：递归 grep

命令行实用工具 grep 最初是作为一个独立应用为 Unix 的一个早期版本编写的，它用于搜索文本文件，找出与所提供的模式(正则表达式)匹配的那些行，并将其打印(在默认情况下)。从那时起，grep(及其各种变种)就成为了每一版 Unix 与 Linux 系统核心库的标准。可以通过各种选项以不同方式改变其行为，包括：

- 递归搜索整个目录子树
- 搜索与模式不匹配的那些行
- 阻止匹配打印
- 打印包含匹配项的文件名(不打印匹配行，也可以打印)
- 打印搜索到的每个文件的匹配数
- 打印匹配的上下文(在匹配前或后几行)

使用流来重现 grep 的行为这个事情本身就很有趣，对于将 grep 的功能嵌入到 Java 程序中也是很有用的(不过请注意，如果真的想要创建一个与 grep 性能相当的替代者，那么你应该多多关注一下本节最后介绍的分割迭代器实现)。首先，对一个目录树中的所有文件执行最简单的 grep 行为，接下来再使用一些选项。

grep –Rh　问题的第一个版本是搜索目录startDir下名字以test开头，并且扩展名为.txt的所有文件，找到其中的小数。这会匹配正则表达式“[-+]?[0-9]*\.?[0-9]+”。我们的程序模拟的是grep -Rh的行为：-R选项会强制grep递归搜索文件系统，-h选项则使用找到的文件名来代替匹配输出行的惯用前缀(下一个问题将会实

现该特性)。我们需要如下的声明：

```
Path start = new File("startDir").toPath();
Pattern pattern = Pattern.compile("[-+]?[0-9]*\.?[0-9]+");
PathMatcher pathMatcher =
   FileSystems.getDefault().getPathMatcher("**/test*.txt");
```

这里用到了两种模式匹配：正则表达式与 globbing，你可能知道，他们用作 shell 模式匹配。globbing 在这里用于展示将其引入到流处理程序中是多么的简单。虽然它的功能没有正则表达式匹配那么强大，但它更为简洁，更为方便。

我们可以将之前介绍管道操作中异常抛出这个主题时所编写的代码作为起始点。方便起见，这里将代码重复一遍，唯一的变化是这里作了一个新的假设，即闭合方法会因无法打开起始目录而重新抛出 IOException 异常，因此代码中无须对其进行处理：

```
try (Stream<Path> paths = Files.list(start)) {
   paths.flatMap(path -> {
      Stream<String> lines;
      try {
         lines = Files.lines(path);
      } catch (IOException e) {
         e.printStackTrace();
         lines = Stream.of("Unreadable file: " + a);
      }
      return lines;
   })
   .forEach(line -> {});
} // no longer catching IOException
```

我们还需要对代码做一些调整。先停下来，仔细检查一下代码，看看为了解决问题需要做哪些变更并做一个列表(我们依然使用 printStackTrace 来处理异常)。

下面是变更列表：

- 调用 Files.list 只会看到 startDir 中的文件。要想搜索目录子树，我们应该使用 Files.walk。
- 出于在异常讨论中所谈及的原因，对延迟计算方法 Files.lines 的调用必须包装在将 IOException 作为 UncheckedIOException 重新抛出的代码中，或是调用 Files.readAllLines。Files.readAllLines会返回一个可作为流源的字符串列表。
- 管道需要对 Files.walk 生成的路径使用一些过滤器：移除来自 Path 流的目录(这要比 readAllLines 的失败捕获更为高效)，确保只处理与名称模式“**/test*.txt”匹配的文件。
- 还需要对文件所返回的文本行应用过滤器：移除对非文本文件处理所产生的空字符串，当然了，还要移除与正则表达式不匹配的行。
- 终止操作应该打印出匹配行。它使用了 forEach，这不会使用排序。现在，假设我们只关注于如何获取匹配行而不是它们出现的顺序。

将这些变更应用到上述代码中，我们就得到了与 grep -h 行为一致的代码：

```
try (Stream<Path> pathStream = Files.walk(start)) {
   pathStream
      .filter(Files::isRegularFile)
      .filter(pathMatcher::matches)
      .flatMap(path -> {
         try {
            return Files.readAllLines(path).stream();
         } catch (IOException e) {
            return Stream.of("");
         }
```

```
        })
        .filter(line -> ! line.isEmpty())
        .filter(line -> pattern.matcher(line).find())
        .forEach(System.out::println);
}
```

grep –R 改变一下这个解决方案，考虑删除选项-h，这样 grep 的每个输出行就会以所找到的文件路径作为前缀。在继续阅读之前，先想一想该如何修改代码来实现该需求，如果必要，则重构。

显然，只有文件行存在时才能将路径作为前缀加上去，flatMap 的 lambda 表达式已经表明了这一点。在内部的 try 体中，lambda 参数 path 在作用域中，可以作为前缀添加到每一行：

```
try {
    return Files.readAllLines(path).stream()
        .map(line -> path + ": " + line);
} catch (IOException e) {
    return Stream.of("");
}
```

这是一种解决方案，不过有些浪费，因为它会对每个文本文件的每一行执行字符串拼接操作(一个代价高昂的操作)，但其中会有很多会被后面的正则匹配过滤器丢弃掉。可以将正则匹配过滤器从现在的下游位置移动到拼接路径与每一行操作之前的位置来轻松避免这个问题。

```
try {
    return Files.readAllLines(path).stream()
        .filter(line -> pattern.matcher(line).find())
        .map(line -> path + ": " + line);
} catch (IOException e) {
    return Stream.of("");
}
```

这个问题的解决方案很简单，只需要对上述代码做小部分修改：

```
try (Stream<Path> pathStream = Files.walk(start)) {
    pathStream
        .filter(Files::isRegularFile)
        .filter(pathMatcher::matches)
        .flatMap(path -> {
            try {
                return Files.readAllLines(path).stream()
                  .filter(line -> pattern.matcher(line).find());
                  .map(line -> path + ": " + line);
            } catch (IOException e) {
                return Stream.of("");
            }
        })
        .filter(line -> ! line.isEmpty())
        .forEach(System.out::println);
}
```

grep -Rc　-c 选项会抑制正常的输出，只会打印每个输入文件匹配的行数。在继续之前先停下来，好好想想该怎么做。

解决方案相当直接，由于输出是一行接一行的，并且来自于包含匹配文本的那些文件，因此每个文件的输出也必须在传递到终止操作之前被收集到单个字符串中。因此，相对于 flatMap 来说，map 才是恰当的操作，每个路径(例如 startDir/foo)都会匹配到诸如“startDir/foo: 3”这样的字符串。可以通过终止操作 count 获取到行数(应用到每个文件的一系列匹配行上)。代码如下所示：

```
try (Stream<Path> pathStream = Files.walk(start)) {
    pathStream
        .filter(Files::isRegularFile)
        .filter(pathMatcher::matches)
        .map(path -> {
```

```
            try {
              long matchCount = Files.readAllLines(path).stream()
                  .filter(line -> pattern.matcher(line).find())
                  .count();
              return matchCount == 0 ? "" : path + ": " + matchCount;
            } catch (IOException e) {
              return "";
            }
          })
          .filter(line -> ! line.isEmpty())
          .forEach(System.out::println);
    }
```

grep -b　-b 选项要求每一行输出以这一行在文件中的位移作为前缀。因此，在如下向这篇“永恒的颂词”中搜索模式“W.*t”：

```
The Moving Finger writes; and, having writ,
Moves on: nor all thy Piety nor Wit
Shall lure it back to cancel half a Line,
Nor all thy Tears wash out a Word of it.
```

会得到如下输出：

```
44:Moves on: nor all thy Piety nor Wit
122:Nor all thy Tears wash out a Word of it.
```

乍一看，这非常类似于第4章介绍的图书位移问题；实际上，我们可以通过一个选项解决这个问题，就像使用自定义收集器那样。不过再仔细分析一下，我们会发现 grep -b 有一个重要的差别，可以提供更棒的解决方案。图书位移问题需要计算不断变化的总数，因此是个前缀和问题，在该问题中，每个元素值取决于前面元素的值。对于针对前缀和的并行算法来说(就像图书位移示例)，组合操作的总代价(即总的延迟时间)与输入数据集的大小正相

关，因此并行化并不会带来我们所期望的性能改进[1]。

不过，我们可以重新思考这个问题来避免困境。如果可以将输入文件的数据看作是内存中的字节数组，那么每一行的位移就都可以确定下来，而无须使用前面的位移。Java NIO 的内存映射特性提供了这个功能：操作系统可以通过它透明、高效地管理文件存储与内存之间的一致性。对于中等或是大型文件来说(一般来说，上百 KB 大小以上的文件才有必要使用内存映射)，内存映射可以带来巨大的性能提升；特别地，随机访问不会有任何性能损失。在 Java 中，内存映射是通过类 java.nio.MappedByteBuffer 实现的。

注意之前的 grep 示例，就像大多数文件处理任务一样，它也不太可能从并行化中获益，这是因为性能瓶颈出现在文件输入阶段，而非流处理阶段。因此，我们将要使用的 Spliterator 技术将会改进这类任务的性能，前提是文件足够大，超过了内存映射的开销。

在该问题中，内存映射还有一个额外的好处，那就是无须计算每一行与其前驱之间的位移；相反，当自定义分割迭代器将缓存划分为行时，我们可以通过其在缓存中的索引得到每一行的位移。分割迭代器的功能就是将值对象提供给流，每个对象都封装了行与其索引：

```
class DispLine {
    final int disp;
    final String line;
    DispLine(int d, String l) { disp = d; line = l; }
    public String toString() { return disp + " " + line; }
}
```

[1] 该问题的严重性取决于组合操作的代价是怎样的。Java 8 提供了 java.util.Arrays，其新方法 parallelPrefix 具有多种重载形式，用于高效计算前缀和，不过这一创新并没有添加到 Stream API 中。

分割迭代器通常会根据要分割的数据结构而命名，不过为了充分描述该分割迭代器，其名称不仅应该包含数据结构 ByteBuffer，还要包含生成的对象类型。由于名称 ByteBufferToDispLineSpliterator 太长了，难以用在打印成书的代码中，因此我们将其简写为 LineSpliterator。LineSpliterator 涵盖了两个索引(lo 与 hi)之间的一系列 ByteBuffer(首尾都包含)，索引会在 LineSpliterator 构建时提供：

```
class LineSpliterator implements Spliterator<DispLine> {
   private ByteBuffer bb;
   private int lo, hi;
   LineSpliterator(ByteBuffer bb, int lo, int hi) {
      this.bb = bb; this.lo = lo; this.hi = hi;
   }
   // implementation of the abstract methods of Spliterator
}
```

在继续之前，我们先来看看实现 LineSpliterator 时需要提供哪些方法，先从 trySplit 开始。

图 5-1 展示了刚开始处理时的缓存。创建的第 1 个分割迭代器的范围包含了整个缓存中的数据，包括终止换行符。

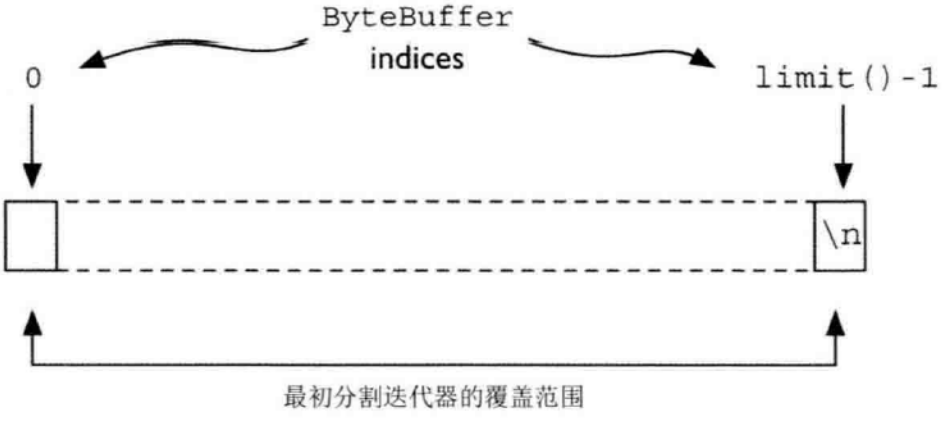


图 5-1 最初的 LineSpliterator 配置

如果要分割这个范围，那么我们的目标是将这个缓存划分为两个大小相近的部分，每一部分都以一个换行符作为终止。我们可以对换行符进行线性搜索，从缓存中间点开始向两个方向进行，从而找到这两个部分之间合适的分割点。该搜索代表了分割开销的算法部分(其余就是创建并派生新的 fork/join 任务的基础设施开销)。其开销与行的平均长度正相关，而非图书位移程序的总缓存大小(参见 4.3.2 节)。

图 5-2 展示了该搜索过程，以及得到的两个分割迭代器的覆盖范围。

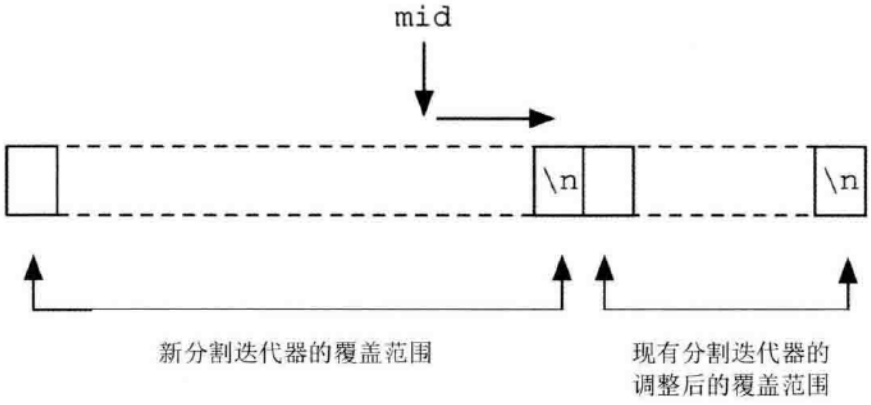


图 5-2　LineSpliterator.trySplit 的操作

假设在遇到 LineSpliterator 范围的结尾时也没有找到换行符。考虑到搜索是从中间位置开始的，因此没有找到就表示整个范围内可能只有几行(也许是一行)，在这种情况下，合理的做法是不要做分割，返回 null，而非沿着相反的方向继续搜索换行符。

理解了 trySplit 的算法后，如下代码就是非常直接的了：

```
public Spliterator<DispLine> trySplit() {
    int mid = (lo + hi) >>> 1;
    while (bb.get(index) != '\\n') mid++;
    LineSpliterator newSpliterator = null;
    if (mid != hi) {
```

```
            newSpliterator = new LineSpliterator(bb, lo, mid);
            lo = mid + 1;
        }
        return newSpliterator;
    }
```

如果没有编写过 tryAdvance 代码，那么请停下来，我们看看该如何实现。

tryAdvance 的目的旨在简化对下一个可用 DispLine 实例的处理。它会搜索下一个换行符，根据搜索遍历的字节创建一个 DispLine 实例，然后将 Consumer 参数应用到该实例上。最后，它会汇聚分割迭代器的范围，排除处理过的字节，并返回一个 boolean 值，表示后续输入的可用性。

```
public boolean tryAdvance(Consumer<? super DispLine> action) {
    int index = lo;
    StringBuilder sb = new StringBuilder();
        do {
    sb.append((char)bb.get(index));
    } while (bb.get(index++) != '\\n');
    action.accept(new DispLine(lo, sb.toString()));
    lo = lo + sb.length();
    return lo <= hi;
}
```

必须要重写另外两个 Spliterator 方法。方法 estimateSize 会返回一个估算值，在该示例中就是重复调用 tryAdvance 所返回的元素数量的近似值。

```
    public long estimateSize() { return (hi - lo +
1)/AVERAGE_LINE-LENGTH; }
```

方法 characteristics 会返回该 Spliterator 的特性，框架会通过它选择恰当的优化策略(参见 6.3 节)。下面是针对该问题的

characteristics 的一个恰当的实现：

```
    public int characteristics() { return ORDERED | IMMUTABLE |
NONNULL; }
```

最后，如下代码展示了如何通过 LineSpliterator 模拟 grep -b 的动作，这里搜索的是单个文件；之前的示例展示了如何将其扩展来执行递归搜索：

```
try (FileChannel fc = FileChannel.open(start)) {
   MappedByteBuffer bB =
        fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());
   Spliterator<DispLine> ls =
        new LineSpliterator(bB, 0, bB.limit() - 1);
   StreamSupport.stream(ls, true)
      .filter(dl -> pattern.matcher(dl.line).find())
      .forEachOrdered(System.out::print);
}
```

5.2.1 节介绍了如何通过 StreamSupport 从分割迭代器创建流。在该示例中，我们为 StreamSupport.stream 的第 2 个参数提供了 true 值来确保生成的是并行流而非串行流。对于终止操作来说，我们使用 forEachOrdered 来生成输出。这与之前的示例不同，那时调用并行化的意义并不是很大。当然了，从原理上来说，forEach 也可以在串行流上以无序的方式执行，不过这只是针对早前的 grep 选项来说的。对于该示例来说，如果匹配行出现的顺序是随机的，那么用户肯定会很奇怪。

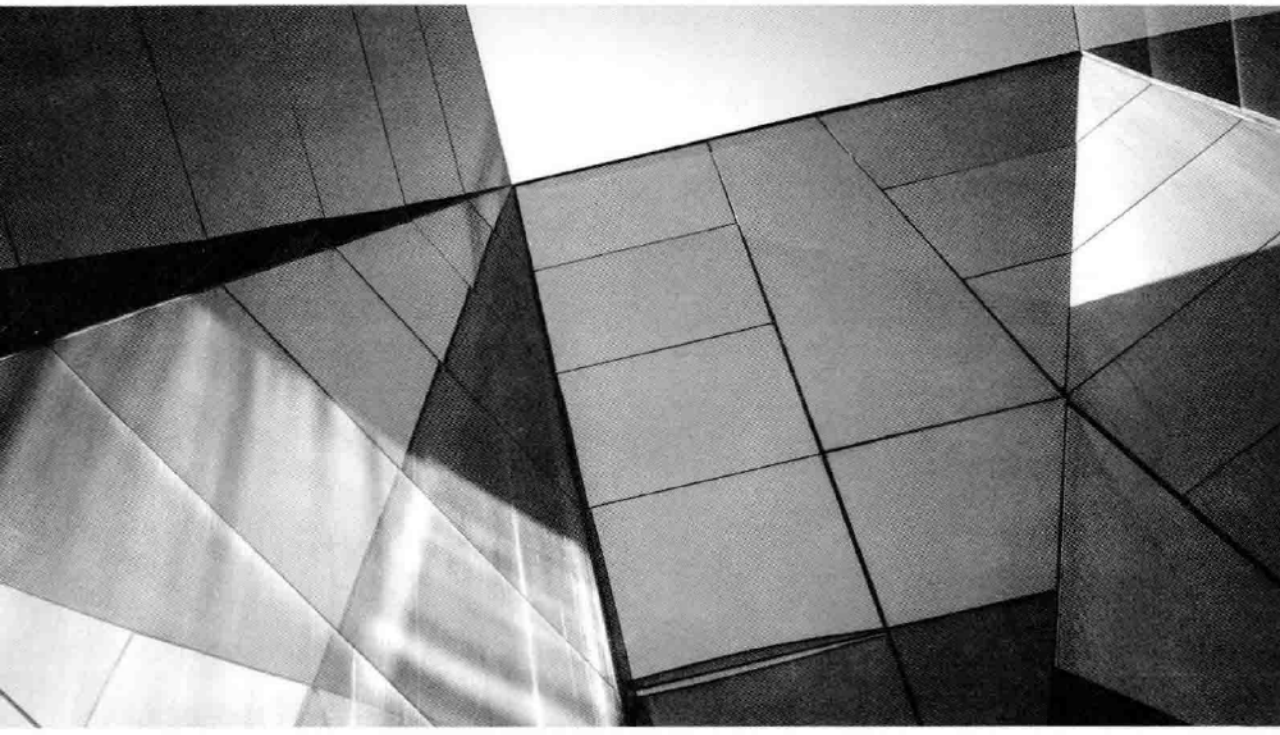
总结一下，我们应该注意到，编写一个功能相当于grep的程序展现出了Stream API的一些优势：之前示例的解决方案都是很直接的，模拟不同的选项只需要对代码做很少的修改。对于最后一个示例来说，我们看到了与Java NIO的搭配使用是如何促使Stream API能够处理大文件输入的，并且还具有非常高的并行化

速度(参见 6.7 节)。

5.5 小结

本章有 3 个目的，且这些目的之间都存在着关联：探索用于创建流的库设施、解释流创建的工作机制，解释如何以及为何要编写自己的实现。我们看到流方法被添加到了众多平台类中，这样流就可以通过中间操作来传递任意种类的数据以进行处理，并最终发送给终止操作。在通过这些方法处理流时，你会受益于并行代码的丰富表现力与可读性；是否可以高效地提取出并行化取决于流源分割数据的效率。不过，分割只是整个过程的一部分，第 6 章将会介绍它是如何与其他因素交织在一起的，例如数据集大小和管道负载等，从而共同确定并行化所能带来的速度提升。

本章给出的主要示例展示了 Stream API 的一些优势：之前针对 grep 的解决方案是很直接的，并且模拟不同的选项只需要对代码做很少的修改即可。在示例的最后一部分中，我们看到与 Java NIO 的联合是如何促使 Stream API 处理大文件的输入，并且并行处理速度也得到了极大的提升(6.7 节将会对此作详细介绍)。



第6章

流的性能

本书提出了使用 lambda 与流的两个原因：编写更好的程序，获得更好的性能。到目前为止，我们看到的代码示例已经展示了通过 lambda 与流可以编写出更好的程序，不过性能方面却并没有给出相关说明。本章将会通过微基准测试技术阐述这一点。

这么做其实是存在风险的：来自于微基准测试的结果可能会被用于以不恰当的方式指导程序设计。由于大多数情况都是不合适的，因此这就是个严重的问题！例如，程序员的直觉对于在复

杂系统中定位性能瓶颈是非常不可靠的；找出性能瓶颈的唯一可信的方式就是度量程序的行为。这样，在一开始开发时，由于并没有可用的系统供我们度量，因此这个时候对系统进行优化就是非常不适合的。

这是非常重要的，因为错误地优化本不是性能关键的代码，实际上是有害的。

- 浪费人力物力，即便是成功的优化也不会改进整个系统的性能。
- 分散对程序设计重要目标的关注，而这个目标是产生结构良好、可读性强、可维护性好的代码。
- 对源代码的优化可能会阻碍即时编译器应用自己的(当前与未来的）优化手段，因为这些优化对于一般性问题都有很好的效果。

也就是说，程序员乐于学习关于代码性能的知识，而且这么做也是好处多多的。首先就是 Stream API 赋予开发者的代码调优能力(通过选择串行执行和并行执行)，这不会改变其设计或行为。这种非唐突的性能管理对于 Java 核心库来说是全新的，不过它会像企业编程那样取得成功并得到广泛应用。要想利用这一点，你需要理解它对于流性能的影响。

不过，即便你从来没有直接使用过该模型，它也会有助于指导设计，对于程序员已经具备的关于传统 Java 代码执行的心智模型会起到补充作用[1]。此外，确实存在一些情况需要我们进行优化。举个简单的例子，假设你已经识别出了一个性能关键的批处理过程，想要在实现的各个选项之间做出抉择。该使用循环还是流呢？

---

[1] 遗憾的是，老生常谈的关于性能的心智模型本身已经过时了；很多程序员并没有意识到 JIT 优化或是诸如管道与多核 CPU 等硬件特性所带来的效果。

如果使用流，那就可以以不同方式将中间操作与收集器组合起来；看看哪种方式最好。本章将通过图6-1所示的结果帮助你回答这些问题：

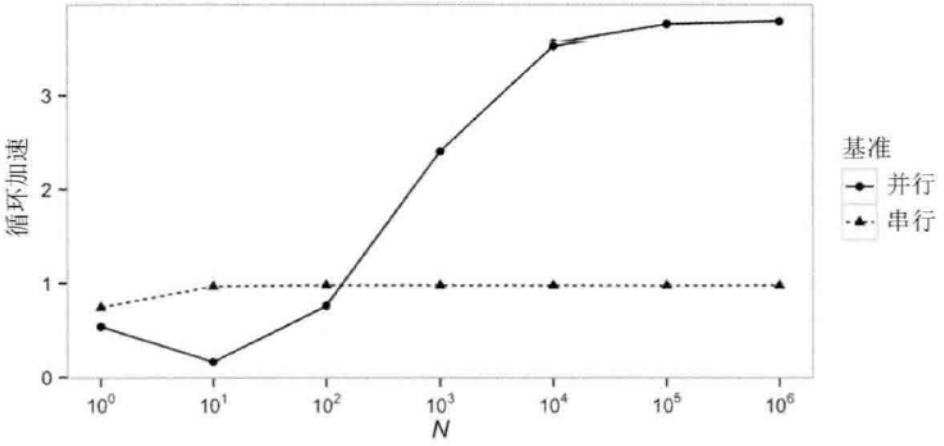


图 6-1 流的性能：串行流与并行流

图 6-1 的结果很有趣，也很有用，不过我们还需要知道更多信息。这张图并没有给出关于结果所产生的实验条件的任何信息。结果可能会被基准外部的因素所影响，如变化的系统负载、其他进程的行为、网络与磁盘活动、硬件、操作系统与 JVM 配置，还有可能被基准本身的属性所影响，如待处理的数据，以及所执行的操作等。

显然，这是个问题；我们希望排除这些因素来讨论流操作的性能。如何通过对复杂系统的简单观察来得到有意义的结果呢？类比一下(可能有点牵强附会)，考虑追踪一款新药对于人体的效果的困难性。试验的目的可能很简单，看看新药对于某种疾病来说是否有效，不过要想得到结果却一点也不简单：对于生病的人来说，你无法管理新药对于他的作用，并观测病人是否康复。许多因素都会影响到结果：病人的健康背景、饮食习惯、性别等。所有这些因素都会影响到个体结果。

当然了，计算机系统远不如生物系统那么复杂，不过对于观测实验来说，它们也是够复杂的了：就像你不能管理病人吃药并观测结果来得出药效的结论一样，你也无法在工作着的系统中修改一行代码，然后期望通过观测结果来得到有用的结论。在这两种情况下充满着太多的变数。下一节将会介绍一种用于消除这些影响的试验性方法。接下来，在本章的后续部分中，我们会应用该方法来度量流操作的性能。

本章的很多试验都来自于之前用于解释流操作技术时所用的示例。这些示例几乎都是用来说明引用流的使用；出于这个原因，并且由于实际上是引用流引发了严重的性能问题，因此本章将会重点关注引用流而非原生流。

6.1 微基准度量

我们的目标是发现具体代码片段的执行代价。不过到目前为止，本章的争论焦点集中在这种度量只有在拥有真实负载的工作着的系统上才是有意义的。分析或是度量生产系统通常都是不切实际的；因此，在负载测试平台上模拟生产系统才是更为可行的做法，但这么做涉及搭建在各个方面都与生产系统一模一样的环境的难题：硬件与操作系统配置、其他进程对资源的需求、网络与磁盘环境，当然还要模拟典型与极端的负载。

更切合实际的做法是微基准度量，它会在隔离(不是真实的)的系统中比较两个不同代码片段操作。按绝对价值计算，每个结果都是毫无意义的，不过二者之间的比较是有意义的——前提是所有这些方面都是可控的。可控环境的想法从原理上来说是非常简单的。如果之前列出的所有因素(硬件与软件环境、系统负载等

等)都保持不变，只有被比较的代码是不同的，那么结果的不同肯定就是由变化造成的。

6.1.1 度量动态运行时

即便可以成功控制对程序行为的外部影响，其他陷阱还是存在的。假设我们的试验遵循如下常见模板：

```
long start = System.currentTimeMillis();
// execute code being benchmarked
long end = System.currentTimeMillis();
System.out.println("My task executed in " + (end - start) + "
ms.");
```

这种模式的简单性将一些严重的困难给隐藏了。其中一些困难与度量过程本身有关，特别是运行次数不多的情况：System.currentTimeMillis可能不如其名字所表示的那样精确，因为它依赖于系统时钟，在某些平台上可能每隔15ms或更少才会更新一次。对于很多平台来说，更好的替代者是System.nanoTime，其更新周期通常是微秒级别的(不过这会增加度量开销)。此外，这些系统调用度量的是经过的时间，因此它们还会包含垃圾收集等开销。

JVM 本身也会对效果造成影响。很多都与字节码到本地代码编译器所施加的优化有关。下面给出 JVM 操作可能会产生的度量问题示例。

热身效应

度量某个代码片段前100次的执行开销常常与1min后再进行同样的度量有着巨大的差别。有很多初始化开销会对一开始的度量造成影响，例如类加载——这就是一个代价高昂的操作，JVM会将其推迟到真正需要这个类时才加载。JIT 编译是另一个不精

确结果的源泉：一开始，JVM 会直接解释字节码，不过当代码执行了足够多的次数后，JVM 会将其看成“热点”，然后将其编译为本地代码(接下来出于未来进一步优化的考量，会继续对其进行分析)。这种编译开销(编译完之后则会带来巨大的性能改进)所产生的结果相比于一成不变的情况来说会高很多。

无用代码消除

在编写微基准度量时，一个经常出现的问题就是要确保获取计算的结果。无用代码消除(DCE)是一项重要的编译器优化技术，其目的在于消除无用代码。例如，如下代码尝试度量加法操作的成本：

```
long start = System.currentTimeMillis();
long sum = 0;
for (int i = 0 ; i < 1_000_000 ; i++) {
    sum += i;
}
long end = System.currentTimeMillis();
System.out.println("Elapsed time: " + (end - start) + "ms");
```

在现代 JVM 上运行该测试会得到非常短的执行时间，这是因为编译器检测到既然 sum 从来没有使用过，那么就没有必要计算它了，因此也没有必要执行循环。对于该情况有个简单的修复可以使用：在输出语句中加入 sum 最终的值就可以确保 sum 得到计算，不过编译器可能还是会检测到它可以通过数学公式计算它而无须遍历；一般来说，如果不引入代价高昂的新操作，我们很难禁止编译器应用 DCE。

垃圾收集

这并非优化的结果，不过却是使用自动化内存管理来度量程序时所固有的问题：几乎所有基准都会创建对象，在基准运行时，

有些对象会成为垃圾。最终，垃圾收集器会被调用，其执行会增加最终的统计时间结果。我们可以通过在度量前手工触发垃圾收集以降低其影响，不过这么做不太现实：事实上，如果会对生产造成很严重的性能影响，那么我们就需要在基准度量中加入垃圾收集的结果。更好的做法则是运行程序足够长的时间，将这些成本分摊到多次迭代中。

6.1.2 Java Microbenchmarking Harness

我们必须控制这些影响以确保度量的结果是我们想要的。控制这些影响是个非常常见但又困难的事情，因此基准框架变得越来越流行。这些框架会创建自定义的测试类，旨在避免微基准度量的各种陷阱。测试类是由框架从你所提供的基准源代码合成而来的。Google Caliper(http://code.google.com/p/caliper)就是个基准测试框架，用于获取本章基准测试结果的框架则是Java Microbenchmarking Harness(JMH)(http://openjdk.java.net/projects/code-tools/jmh)。

JMH 有助于解决上一节列出的各种问题。例如，JMH API 中包含了一个静态方法 Blackhole.consumeCPU，它会与参数成比例地消耗 CPU 时间，同时又没有副作用，没有 DCE 和其他优化的风险(所用的绝对时间量并不重要，因为 consumeCPU 只用于微基准的各种比较)。

要想通过JMH创建基准，你只需要在方法上加上注解@Benchmark。JMH有不同的使用模式；在吞吐量模式下，它会根据你所指定的时间周期重复执行每个被注解的方法，自动将返回值传递给方法Blackhole.consume以阻止DCE。例如，下面是生成本章一开始的图的3个方法之一。在该示例中，返回值总是一个空的Optional，不过将其返回意味着VM必须要对每个流元素计算出

过滤器predicate：

```
@Benchmark
public Optional<Integer> sequential() {
    return integerList.stream()
        .filter(l -> { Blackhole.consumeCPU(payload);
                       return false; })
        .findFirst();
}
```

6.1.3 试验方法

最后，我们为试验结果应用两种标准的试验科学的保障措施。

统计

给定我们知道的会影响微基准度量的因素数量，我们无法期望相同的试验、重复的次数，会得到完全相同的结果；不同的外部因素的影响会随着观测的不同而发生变化。我们可以对最可能的值计算出一个点估算值(例如平均值)，不过如果外部因素主导着试验，那么这个结果就没什么意义了。如果假设外部因素是随机变化的，那么多次运行相同的试验就会增加我们对结果的认可度。这可以通过置信区间(CI)得以量化，置信区间指的是给定一组值，我们可以确保在一定程度上(通常是 95%)“真”值会落在这个区间中，即从多次试验中都可以得到这个值。试验规模越大，我们所得到的置信区间范围就越小。

置信区间是试验意义的一个很有用的指示器：如果两种情况下的置信区间没有重复，那就表示这些情况会产生不同的结果，我们总是会假定其他因素的变化是随机的，不会偏向于其中任何一种情况。实际上，对于本章的试验来说，相比于方式上的差异，条件之间的差异是很小的，试验次数也是足够多，置信区间实际上太小了，无法反映出来。

开放同行评审

就像所有的科学试验一样，性能度量也是存在着瑕疵和偏差的。对于这些错误源来说，一种安全措施就是对度量实验进行开放同行评审。这意味着我们不仅要公开汇总结果，还要公开关于试验条件的足够详尽的信息，让其他人能够重现并检查。这些信息包括硬件、操作系统与JVM环境，还有关于结果的统计信息。接下来是一个示例，即图6-1的基准及结果数据。由于重复本章的每个试验信息实在是太过于单调乏味，因此本书网站的URL给出了全部的配置信息；因为这些试验很容易重现，因此这里并没有给出原始试验结果。

```
    @State(Scope.Benchmark)
                // The objects of this scope (e.g., instance
                // variables) are shared between all threads.
    public class CompareByN {

        // entire benchmark is run for each value of the @Param
annotation
        @Param({"1", "10", "100", "1000", "10000", "100000",
"1000000"})
    public int N;

    private final int payload = 50;
    private List<Integer> integerList;

    @Setup(Level.Trial)
    public void setUp() {
        integerList = IntStream.range(0,
N).boxed().collect(toList());
    }

    @Benchmark
    public void iterative(Blackhole bh) {
        for (Integer i : integerList) {
            Blackhole.consumeCPU(payload);
            bh.consume(i);
```

```
    }
}

@Benchmark
public Optional<Integer> sequential() {
    return integerList.stream()
        .filter(l -> {
            Blackhole.consumeCPU(payload);
            return false;
        })
        .findFirst();
}

@Benchmark
public Optional<Integer> parallel() {
    return integerList.stream().parallel()
        .filter(l -> {
            Blackhole.consumeCPU(payload);
            return false;
        })
        .findFirst();
    }
}
```

JMH 用于运行这些基准并使用默认设置，对于每个 limit 值来说，它会启动一个新的 JVM，然后针对每个注解的方法执行如下步骤(称为“迭代”)：

(1) 对于热身来说，它会重复执行方法 1s，20 次。

(2) 对于度量来说，它会再次重复执行方法 1s，20 次；这一次，它会记录每秒运行时方法的调用次数(即“分数”)。

表 6-1 的每一行表示对于给定的数据集大小，20 个迭代采样的平均执行数与置信区间。

虽然简单，但这张表格展示了 Intel Core 2 Q8400 Quad CPU(2.66 GHz)、2MB L2 缓存上运行的 Linux 系统(内核版本 3.11.0)，JDK1.8u5 上执行基准的结果。“错误(99.9%)”一列展示

了99.9%置信区间中值的分布(99.9%是JMH默认的置信区间，对于正常的统计标准来说是很高的)。

表6-1 图6-3的基准的采样结果

基　准	参数N	分　数	分数错误(99.9%)
CompareByN.iterative	1	435668.6	429.0
	10	43001.1	2.9
	100	4316.4	175.8
	1000	432.3	0.1
	10000	43.2	0.2
	100000	4.3	0.0
	1000000	0.4	0.0
CompareByN.parallel	1	390068.6	769.7
	10	33339.6	406.2
	100	10674.4	128.5
	1000	1556.4	8.2
	10000	168.1	1.2
	100000	16.8	0.4
	1000000	1.6	0.0
CompareByN.sequential	1	412777.1	1053.1
	10	42967.8	6.3
	100	4286.6	8.9
	1000	428.0	3.7
	10000	43.6	0.0
	100000	4.3	0.0
	1000000	0.4	0.0

本章后续部分会以图 6-1 的形式展现微基准测试的结果，但不会提供这些细节层次。不过重要的是，你已经知道了一切都是可以得到的：每个试验的完整条件、用于准备展示的统计方法，以及必要情况下所需的原始结果。这样就可以评审试验的方法了，如果必要还可以自己重做，最为重要的是，可以设计并执行自己的度量试验。

在阅读后面的内容时应该给自己提个醒：硬件与 Java 平台的发展迟早会改变这里讨论的很多权衡情况，通常都要支持并行。就像对性能的具体方面的所有讨论一样，这部分资料也只是之前的经验而已，说不定什么时候就过时了！

6.2 选择执行模式

本书已经介绍过，并行化是流编程的一个主要推动力。不过到目前为止，我们还没有谈及何时应该利用这一点[2]。有3组因素值得我们考虑。

执行上下文

程序执行的上下文是什么？记住，并行流是通过普通的 fork/join 池实现的，在默认情况下，它会根据操作系统获得的硬件核心数目来确定线程数。操作系统所返回的结果并非一成不变的；例如，可能会考虑到超线程(可以让两个线程共享一个核心)，这样当一个线程在等待数据时，核心就不会空闲下来。在受处理器限制的应用中，这会导致在同一时刻并行化的线程比实际可用

[2] 回忆一下 3.2.3 节，所有流都公开了方法 parallel 与 sequential，它们可以设定整个管道的执行模式。当时曾指出，如果在一个管道上调用这些方法多次，那么最后一次调用将会决定执行模式。

的线程多的结果。

执行上下文的另一方面就是会与运行在同一硬件上的其他应用竞争。普通的 fork/join 池配置的一个隐式假设就是当并行操作执行时，对于硬件没有其他的需求。如果机器已经处于高负载的情况，那就可能出现一些 fork/join 池线程非常需要 CPU 时间的情况，还有可能出现其他应用性能降低的情况。

如果这些问题意味着你需要减少普通的fork/join池中线程的数量，或是由于其他原因要对其进行更改，那就可以设置系统属性java.util.concurrent.ForkJoinPool.common.parallelism来重写fork/join池的默认配置。当然，如果设置了该属性，那么就需要通过度量应用性能的变化来证明修改是正确的！

除了外部竞争，在确定是否要并行执行流时你还应该考虑内部竞争。有些应用已经是高并发的了；例如，考虑一个 Web 服务器，如果能将每个用户请求分配到单独的核心上执行，那么其性能就处于最佳状态。这个策略显然与试图接管所有核心以执行单个请求是不同的。

工作负载

5.2节曾介绍过，创建并行流需要很多机器。这么做所引入的延迟需要通过对流元素的同时处理所节省下来的时间来平衡。如果不使用并行，那么N个任务的执行时间(每个任务需要Q时间)就是NQ。如果在P个独立的处理器上执行并行，那么时间就是NQ/P。这两个时间的差别必须要大于并行的开销。显然，N与Q越大，效果越明显。图6-3展示了在3种不同的Q值下，对于一个纯粹的处理任务来说，串行流与并行流的相对执行速度。操作成本Q是由JMH方法Blackhole.consumeCPU提供的。其值在不同的机器上是不同的；该试验是在之前提及的Q8400机器上执行的，在上面Q值大约

为5ns。图6-3展示了并行与串行执行的损益平衡点(线的交叉处)，其中NQ大约为40ms，这大概要比Oracle团队给出的范围低5倍。图6-2针对本书第1个流程序(1.2节介绍的“maxDistance”问题)给出了一个类似的NQ响应。

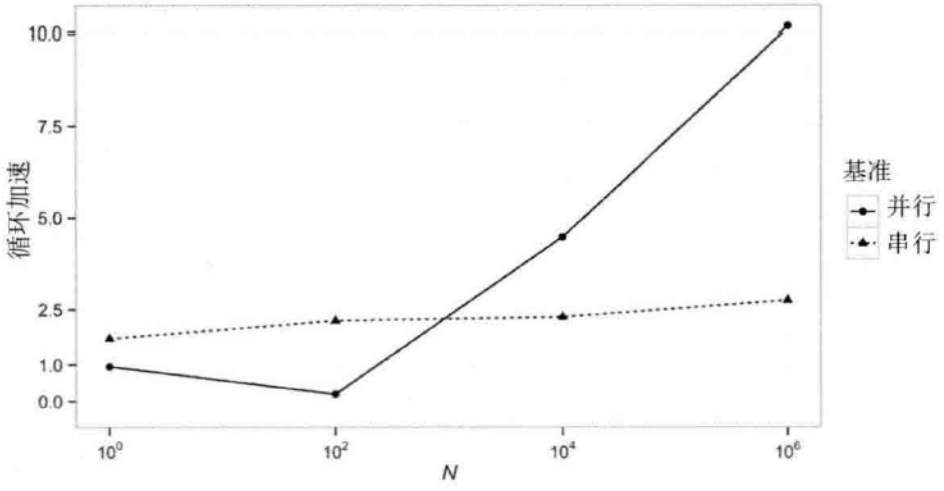


图 6-2 "maxDistance"程序性能(http://git.io/r6BtKQ)

对于并行执行来说，P 的最优值并不是绝对的，而是要取决于负载。如果 NQ 不太大，那么分解负载的开销就会比并行化带来的收益大。负载越大，增加处理器数量所带来的提速就越明显(这个非正式表述来自于 Gustafson 法则，它重新构建了知名的 Amdahl 法则，并且得出了不那么悲观的结论。与 Amdahl 法则(它对并行加速设定了非常紧的限制，前提是使用固定的 N 与可变的 P)相反，Gustafson 法则设定 N 会随着 P 的变化而变化，因此给出了一个常量运行时间(参见 http://www.johngustafson.net/pubs/pub13/pub13.htm))。

分割迭代器与收集器性能

6.7 与 6.8 节将会对此进行更为详尽的介绍。现在，我们只要注意到当 Q 减少时，通过源进行分割以及通过收集器进行积聚会

变得更加重要；高Q值会导致中间操作成为管道瓶颈，使得并行化更具价值。与之相反，当Q值降低时，流源与终止操作的并行性能在确定是否进行并行时就变得尤为重要了。

应用该模型的一个问题在于估算Q在实际情况下的值是非常困难的，以及具有高Q值的流操作不太可能像试验中那样简单，正如我们在第4章与第5章的示例中看到的那样。如果实际情况如此，那么度量中间管道操作的成本就是非常直接的事情了，试验的结果也是很有价值的；不过在实际情况下，更加复杂的中间操作通常会涉及一些I/O与同步，这会导致从并行化获益变少。与往常一样，如果怀疑，那么请自己重复该试验，修改它以使之适合你的需求。

6.3 流的特性

上一节简单描述了是否要并行的问题。给出的答案则是忽略不同流的行为以及不同的中间操作之间的差别。本节将会深入探究流的属性，后续章节则会介绍不同操作是如何借助于这些属性来选择最佳的实现策略，或是在某些情况下被这些属性所限制，转而采用次优策略。例如，在并行处理的过程中维持元素的顺序对于某些中间操作与终止操作来说会带来严重的性能开销，如果已经知道流的元素是无序的(例如，这些元素来自于HashSet)，那么流操作就可以不保持元素的顺序，从而获得更好的性能。如图6-3所示。

流通过特性公开了此类元数据，这些元数据的值要么为true，要么为false；在该示例中，操作可以检查流的特性ORDERED是否为true。一个流S的特性最初是由其源定义的(事实上，流的特性

是作为Spliterator接口的字段定义的)；随后，对S的中间操作所生成的新流的特性可能与S相同，也可能删除或是增加了某些特性(取决于操作本身)。例如，下面列出了管道中每一阶段的特性。

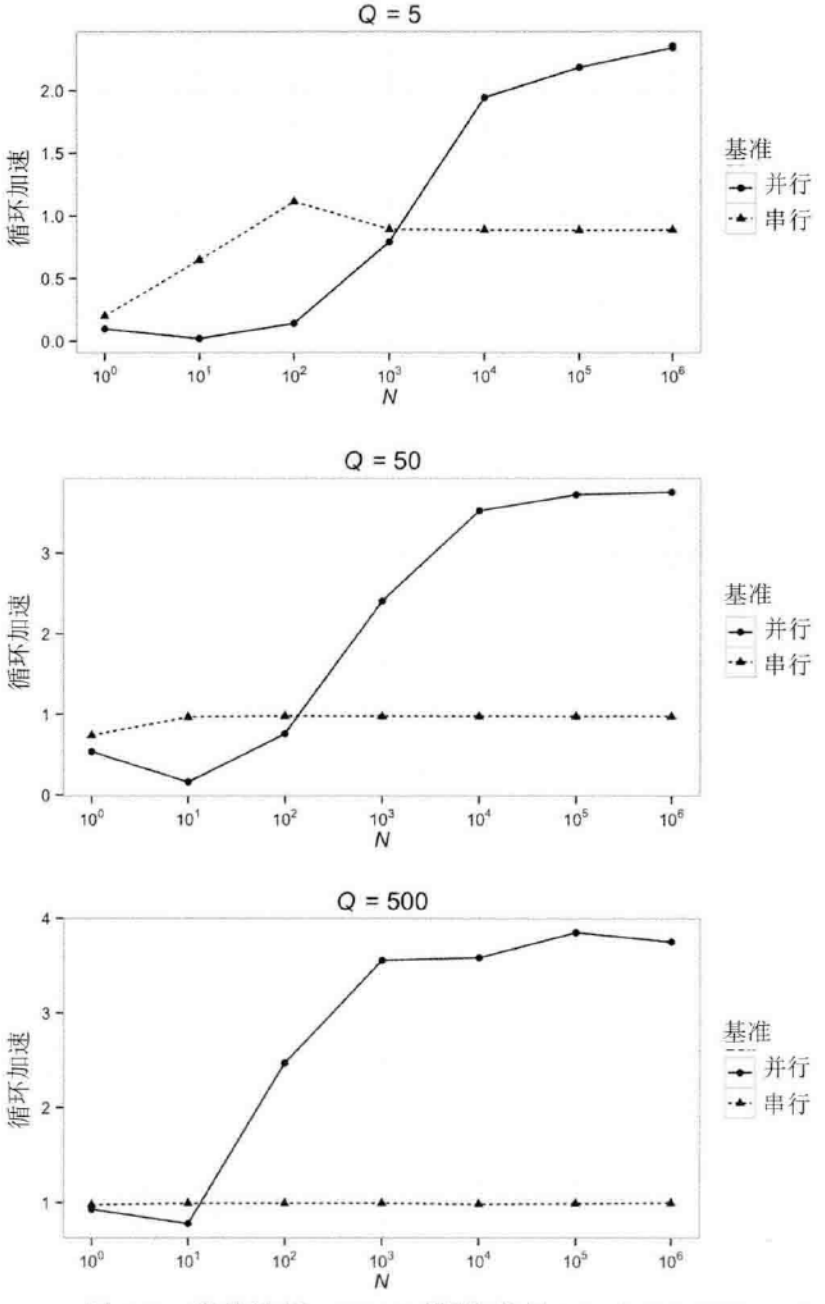


图 6-3　流的性能：N×Q 模型(参见 http://git.io/4NIuqw)

```
Stream.of(8, 3, 5, 6, 7, 4)    // ORDERED, SIZED
    .filter(i -> i % 2 == 0)   // ORDERED
    .sorted()                  // ORDERED, SORTED
    .distinct()                // DISTINCT, ORDERED, SORTED
    .map(i -> i + 1)           // ORDERED
    .unordered();              // none
```

下一节将介绍 ORDERED，本节先来介绍其他特性的含义：

- SORTED：具有该特性的流中的元素是自然排序的，也就是说，其类型实现了 Comparable，它们已经通过其 compareTo 方法排好序了。我们可以根据自己的需求来定义 Comparator，这样流元素就会按照我们所指定的顺序排序，不过这样的流就没有 SORTED 特性了。例如，源为 NavigableSet 的流会具有特性 SORTED，前提是通过默认构造方法而非接受 Comparator 的构造方法来创建其实现。与之相反，源为 List 的流就没有该特性。

  该特性的一个使用示例就是排序：如果操作对象是具有特性 SORTED 的流，那么按照自然顺序进行排序的操作性能就会得到极大的优化。再举一个示例，串行、有序流的 distinct 操作可以通过使用一个简单算法来避免存储元素：只有当元素与其前驱不相等时才将其添加到输出流中。

- SIZED：如果流中的元素数量确定并且已知，那么流就会具有该特性。对于源是非并发集合的所有流来说都具有该特性。它们在流处理的过程中是不会被修改的，因此其大小也是已知的。与之相反，并发集合在流处理过程中可能会插入元素或是删除元素，因此在流处理完毕之前，其元素数量是无法精确预知的。对于大多数源不是集合的流来说(参见 5.1 节)，它们一般都没有该特性。

利用 SIZED 的一个优化示例就是将元素积聚到集合中；如果通过恰当的大小创建集合而非动态改变集合大小，那么其性能就会更好。

- DISTINCT：具有该特性的流中不会存在这样两个元素 x 与 y，使得 x.equals(y)。源为 Set 的流具有该特性，但源为 List 的流则没有该特性。

  例如，如果某个流具有该特性，那么操作 distinct 就会得到极大的优化。

6.4 排序

在上一节所列出的特性中，排序对于性能来说是最为重要的，这是因为对于一些管道，如果能够免掉排序，那就可以极大提升并行流的效率了。此外，正如第 1 章所强调的那样，从我们长期使用迭代处理的经验来看，很多时候都没必要使用排序。回忆一下 1.1.1 节的示例，该示例要求我们按照字母表顺序将信件投递到信箱中。因此，我们有必要区分排序确实非常重要的情况。

如果元素的顺序从语义上来说很重要，那么流就会有一个前后顺序。就拿 List 来说，我们要通过一系列操作将元素添加到 List 中，其契约指定了应该如何存放这些元素，并确保在返回时(例如迭代)保留其顺序。现在，考虑使用流来处理 List 中的元素：

```
String joined = stringList.parallelStream()
    .map(String::toUpperCase)
    .collect(Collectors.joining());
```

如果字符串 joined 无法反映出 stringList 的顺序，那么你肯定会觉得很奇怪和失望。你可以将前后顺序看作是一个属性，流通

过它来保存传递给它的元素的“位置”。对于该示例来说，由于List具有前后顺序(数据结构也有该属性)，因此流亦如此，同时元素的位置也会被保存到终止操作中。例如，如果stringList值为["a", "B", "c"]，那么我们就可以保证joined一定是“ABC”，而非“BAC”或是“CAB”。

与之相反，如果流的源没有前后顺序(例如HashSet)，那就没有元素的位置需要保留下来：流的源没有前后顺序，流当然也没有了[3]。由于流已经显式确定为是并行的了，因此我们就要假设集合中的内容可能会被不同处理器上的不同线程所处理。我们没有办法确保处理的结果(会被joining方法使用)与流开始时的顺序保持一致。如果后续对这个拼接字符串的使用不要求顺序(例如，你只是通过它来得出每个字符的出现次数)，那么指定顺序就会对性能造成影响，并且与程序的目的产生矛盾。

使用这个一般规则并不要求你理解流类实现的细节信息。例如，你可能觉得将一个流置为无序并不会改进程序的性能，如下代码所示：

```
long distinctCount = stringList.parallelStream()
    .unordered()
    .distinct()
    .count();
```

但实际上，对于大规模数据集来说，这么做性能会提升50%左右(http://git.io/i6frOw)。其中的原因就是distinct的实现会测试之前已经存在的元素之间的关系，如果流是无序的，那就可以使用并发Map。不过这个实现细节可能会在未来发生变化，但你应

[3] 注意，虽然你可能觉得所有的Set实现本质上都是无序的，但一个集合的前后顺序却取决于规范是否定义了迭代顺序：在Set的集合框架实现中，LinkedHashSet以及Set子接口SortedSet与NavigableSet的实现就是如此。

该注意到排序，并且只在必要的情况下才使用它，这一原则是不变的。

一开始，术语“前后顺序”与“时间顺序”是相反的，后者用于描述这样一种情况：流中的元素会根据它们生成的时间来排序。既然该情况被描述为是“无序”的，因此就不能通过流没有顺序这句话来表述其思想，而是要说你不能依赖于其顺序。我们很容易就可以看出并行流的时间顺序与前后顺序之间没有必然的联系，例如，观察在管道操作中改变线程安全对象的结果。如果串行执行，那么如下代码：

```
AtomicInteger counter = new AtomicInteger(1);
IntStream.rangeClosed(1,5)
    .mapToObj(i -> i + ":" + counter.getAndAdd(1) + " ")
    .forEachOrdered(System.out::print);
```

通常会产生如下输出：

```
1:1 2:2 3:3 4:4 5:5
```

当并行执行时，我们就无法预测其顺序了：

```
AtomicInteger counter = new AtomicInteger(1);
IntStream.rangeClosed(1,5).parallel()
    .mapToObj(i -> i + ":" + counter.getAndAdd(1) + " ")
    .forEachOrdered(System.out::print);
```

可能会生成如下输出：

```
1:2 2:1 3:5 4:4 5:3
```

但请注意，无论哪种情况，你都不能依赖于它的顺序。

6.5 有状态操作与无状态操作

从直觉上来看，某些中间操作要比其他操作更容易并行化。例如，map 到流这个操作显然可以通过将 map 的行为型参数逐一映射到流元素上来实现。与之相反，sorted(对流元素进行排序)就得等到将流元素的所有值都积聚后才能完成。由于 map 可以在不引用其他元素的情况下处理每一个流元素，因此我们说它是无状态的。其他无状态流操作还包括 filter、flatMap，以及会生成原生流的各种 map 与 flatMap 的变种。只包含无状态操作的管道可以一次性得到处理，无论串行还是并行，只需要很少的缓存，或者根本就不需要缓存。

诸如 sorted 这样的操作(需要记录关于已经处理过的值的信息)称为有状态操作。有状态操作会将一些，甚至是全部流数据(如果需要处理多次)存储到内存中。其他有状态操作还包括 distinct 与流截断操作 skip 和 limit。库实现会通过各种策略在可能的情况下避免内存缓存；这使得预测一个有状态操作会对并行速度产生多大影响变得很困难。例如，Integer 流上 sorted 性能的基准度量可能会给出这样的结果：在 4 核 CPU 上的并行加速为 2.5 倍(http://git.io/6kdkDw)，这说明了优化是正确的。但另一方面，Oracle 团队在设计高效并行 limit 时却遭遇到了意料之外并与直觉不符的困境，这是在并行流中需要格外小心的一个操作。幸好，对流库的进一步工作(实现进一步的优化，并改进现有的操作)将会持续不断地改进这一点，以期望为库的用户提供更好的结果。

6.6 装箱与拆箱

引入原生流的一个重要动机就是要避免装箱与拆箱的开销。3.1.2 节给出了一个操作数字流的简单任务。一个程序使用了原生流：

```
OptionalInt max = IntStream.rangeClosed(1,5)
    .map(i -> i + 1)
    .max();
```

另一个使用了包装值的流：

```
Optional<Integer> max = Arrays.asList(1,2,3,4,5).stream()
    .map(i -> i + 1)
    .max(Integer::compareTo);
```

图 6-4 展示了在不同流长度的情况下这两个程序的性能比较，这两段程序都完成了类似的迭代工作。正如我们所预料的，没有装箱的代码性能超过了装箱的代码；对于大型数据集来说，速度上的提升会达到一个数量级。

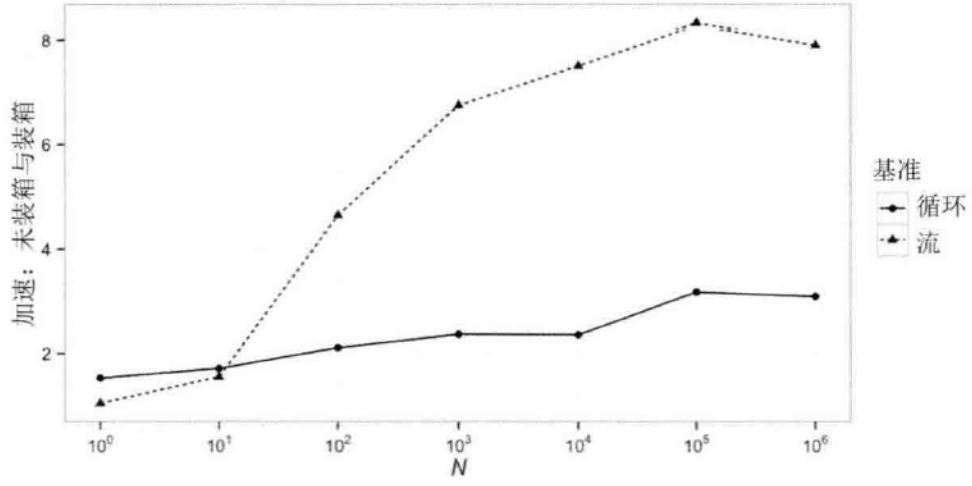


图 6-4 未装箱与装箱的性能比较(http://git.io/YB9V6g)

6.7 分割迭代器性能

流的源对于串行流的性能影响是相对直接的：如果源能够快速提供值，保证流处理的数据供给，那么它对性能的影响就是微乎其微的。如果不是这样，那么它就会成为整个处理的瓶颈。对于并行流来说，问题就变得更为复杂了，因为我们需要两个补充过程(分割与迭代)来解耦源数据并提供给并行线程。一个数据源可能为一个过程提供了非常快的实现，但却无法高效支持另一个过程。例如，如果 UnaryOperator 很高效，那么 IntStream.iterate 就能以非常快的速度处理值，但由于要知道前驱是什么才能计算每一个值，因此它对于分割来说几乎没有什么帮助，这样将其作为源的并行流相对于串行执行来说就不会有速度上的提升。对于常见的流源来说，分割最为高效的是数组、ArrayList、HashMap 与 ConcurrentHashMap，效率最低的是 LinkedList、BlockingQueue 实现以及基于 I/O 的源。

流 I/O 方法以及 iterate 这样的方法有时会从并行化中获益，如果插入了额外的处理步骤，那就会预先将其输出缓存到内存数据结构中，就像之前介绍的 BufferedReader 那样。总结一下，我们会得出这样的结论，如果数据源在内存中，使用了 Stream API 的程序就会从并行加速中获得最大收益，随机访问数据结构有着高效的分割性能(当然，这并不是说拥有其他数据源的程序不会以其他方式从 Stream API 中获益)。

第 5 章中模拟 grep -b 的程序能够很好地说明这一点。将顺序访问的文件读取到一个 MappedByteBuffer 中就非常适合于使用并行。有个试验度量了该程序的性能(http://git.io/-CziKQ)，它比较了使用 3 种不同算法将 MappedByteBuffer 分割为行的速度：迭代、

流串行与流并行。创建 MappedByteBuffer 的开销从度量中排除掉了，从而使我们可以关注于分割算法的效率。结果与数据集大小并没有什么关系：对具有 10 000～1 000 000 行的文件来说，串行流处理的速度是迭代处理的 1.9~2.0 倍；对于同样的规模，并行处理的速度是迭代处理的 5.2~5.4 倍。对于 5.4 节介绍的高效分割算法来说，这个结果没什么好奇怪的。

6.8 收集器性能

就像流源一样，终止操作的行为在并行流下要比串行流复杂得多。对于串行来说，集合只包含了调用收集器的积聚函数，如果能从流中快速使用值，那么它就不会成为整个过程的瓶颈。对于并行流来说，根据是否将自身声明为 CONCURRENT(收集器以特性的方式公开自己的属性)，框架对待收集器的方式是不同的。并发收集器可以确保其积聚操作是线程安全的，这样框架就无须像对非线程安全的结构那样负责并发的积聚调用了。从原理上说，如果框架无须管理线程之间的限制情况，那么并发性能会得到很大的提升。这是可以实现的，正如我们看到的那样，通过选择恰当的数据集与配置参数来实现。

除了 CONCURRENT 外，收集器只公开了两个特性：收集器可以拥有 IDENTITY_FINISH 特性，它告诉框架无须为完成函数提供任何信息；还有一个 UNORDERED 特性，它告诉框架无须维护流中元素的前后顺序，因为收集器会将其丢弃掉。

6.8.1 并发 Map 的合并

库所提供的并发收集器是从 Collector 方法 toConcurrentMap

与 groupingByConcurrent 的各种负载下获取的。使用了框架提供的 ConcurrentMap 实现的收集器(而不是自己提供 ConcurrentMap)都依赖于 ConcurrentHashMap(CHM)。这里需要解释一下这些收集器的性能特性，这些收集器的所有潜在用户都应该理解这些特性。该基准(http://git.io/YXyvpg)表明对于任意数据集来说，groupingByConcurrent 的性能要比 groupingBy 差。分析表明 groupingByConcurrent 是因为对 CHM 的大小频繁修改而导致速度变慢的；使用一个足够大的初始值来创建 CHM 以避免改变其大小，这可以提升约 30%的性能，甚至对于只包含 1000 个元素的数据集来说亦如此。如果比较是公平的，也就是说，如果 groupingBy 所用的 HashMap 的初始大小也随之增加，那么依然不建议使用 groupingByConcurrent，因为改变大小也是 HashMap 的重要开销。

现在来到了我们熟悉的领域，更慢的过程(在该示例中就是将元素添加到CHM与添加到HashMap中相比)对于大规模数据集来说，它会被并行执行下更慢的操作所补偿。如果事先选择了负载因子为 0.5 的大小(http://git.io/HAiaQQ)，那么其相对性能就表明随着数据大小增加到了 100 000 个元素以上，其并发性能也得到提升(如图 6-5 所示)。

事实上，图 6-5 依然没有给出全貌。串行的 groupingBy 会创建新的对象，因此垃圾收集就会成为影响性能的一个因素。Parallel GC 会影响试验结果，它更偏爱串行 groupingBy，这是因为串行操作没有使用的核心还可用于运行并行 GC，这将其代价隐藏了。使用 4 个度量线程运行相同的测试(为 JMH 应用选项-t 4)会在 10 000 个元素内达到损益平衡点。

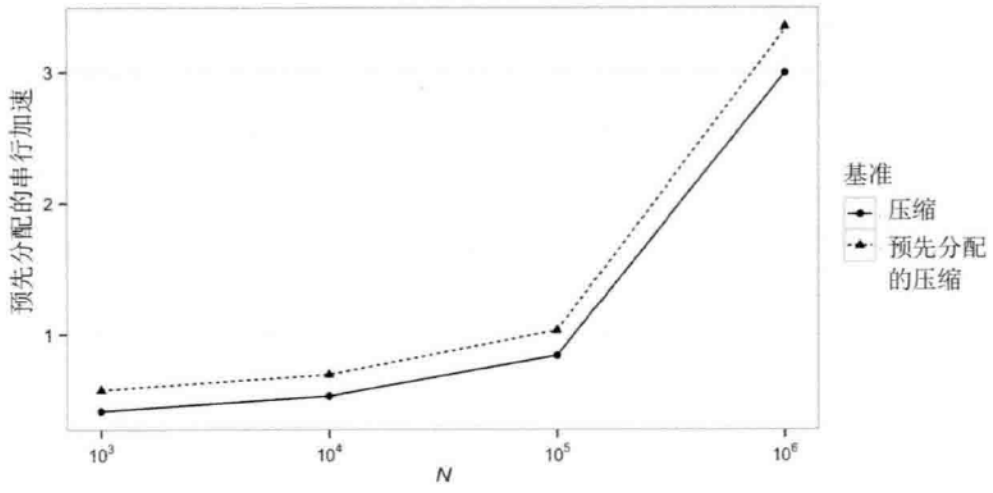


图 6-5 groupingByConcurrent 与 groupingBy 比较

6.8.2 性能分析：对点进行分组

4.3 节介绍的“点分组”收集器是说明并行化的很好的示例；虽然创建的数据结构是一系列非线程安全的 ArrayDeques，不过对其进行的所有收集器操作都是快速的，没有使用迭代，因此在集合处理过程中是没有什么可争论的。如果顺序执行，那么 4.3 节的程序(http://git.io/XdvZ3w)的性能与相应的迭代版本就非常类似；对于 4 核机器上没有什么有效载荷的流来说，并行执行的速度比串行执行要快大约 1.8 倍，对于每个元素有 100 个 JMH 令牌的有效载荷的情况来说，速度大约要快 2.5 倍。

6.8.3 性能分析：找到我的书

最后一个要分析的收集器程序来自于4.3.2节：给定前面图书的页数，该程序会找到每本书在书架上的位置。6.8.1节对并发Map的合并进行了分析，我们应该知道，将该收集器(用于生成Map)用于并行流的终止操作时必须要十分小心。实际上，该程序

(http://git.io/aMoy6w)要比相应的迭代版本慢40%左右。有几个因素造成了这个结果，在实际情况下，每个因素都会导致结果变慢：

- 这是前缀和的问题，其中每个元素值都取决于前面元素的值。对于前缀和的并行算法来说(例如这个)，合并操作的总成本(即总延迟)与输入数据集的大小正相关，无论并行级别是什么都如此[4]。这个问题的严重性取决于合并操作的成本；在该示例中，将右侧 Deque 加入到中间 List 中会降低速度，对于中等工作负载来说，这要比在合并到左侧前对右侧 Deque 执行简单的迭代更新慢 20%左右。
- 完成器会将结果积聚到 ConcurrentHashMap 中，其初始大小目前是默认值 16。对于真实大小的数据集来说，这会导致大量的更改大小操作，这对于 ConcurrentHashMap 来说成本很高，并且会产生大量垃圾——事实上，垃圾收集成本是非常高的。因此，性能度量并不会孤立地给出关于性能的指标。预先指定大小的 CHM 对于性能改进来说作用非常大。即便如此，收集到 Map 中也是个成本高昂的操作；如果指定了一个线性数据结构来持有结果，省略掉了完成器，那么相比于迭代处理来说，并行速度会提升 40%左右。
- 第 4 章介绍的这个程序在收集前并没有对 Book 对象做任何处理。这样，并行流处理就难以发挥作用了——在实际情况中，收集前通常都要进行预处理。对于所有的并行流程序来说，可以使用 fork/join 池执行中间操作的地方越多，并行速度的提升就会越明显。

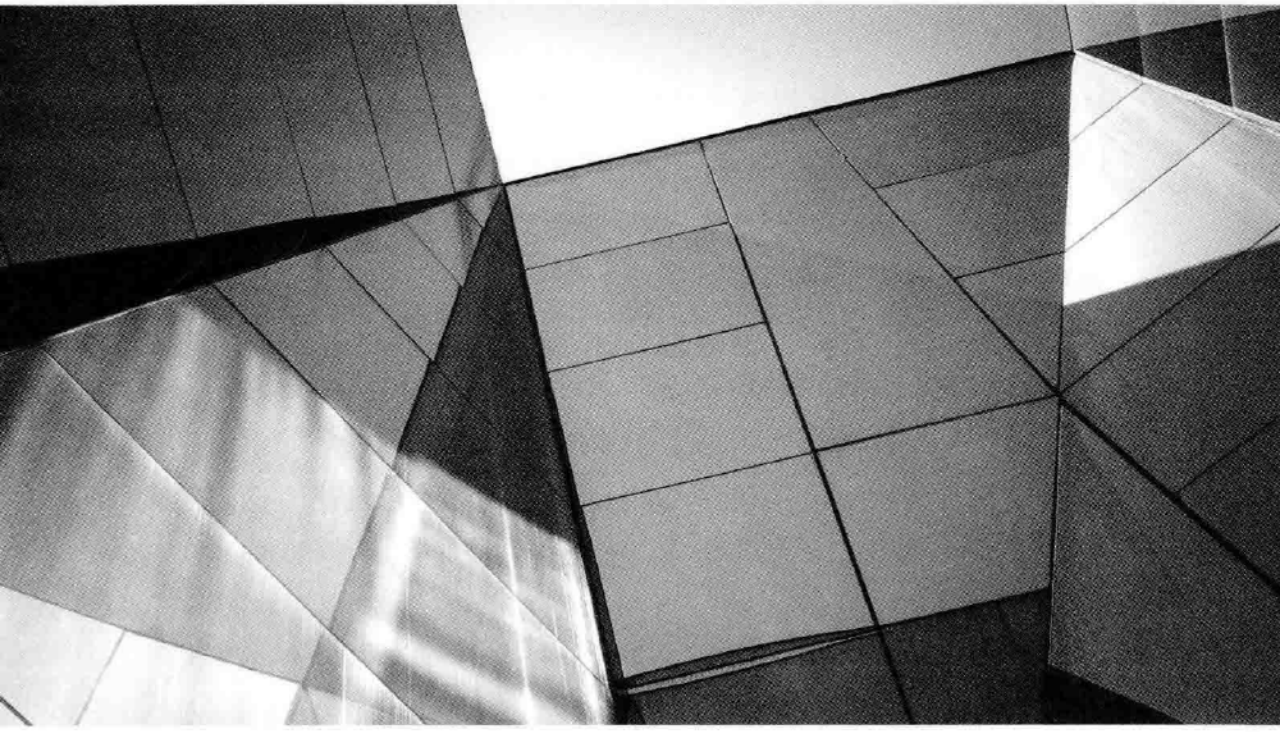
应用了这些改变后(http://git.io/wZe_tg)，并行流程序相比于串

[4] Java 8提供了带有各种负载的 java.util.Arrays 和一个新方法 parallelPrefix，用于高效计算前缀和，不过 Stream API 中尚未加入这些内容。

行版本来说速度快了 2.5 倍(4 核机器，100 万个元素的数据集，每个都需要 200 个 JMH 令牌来处理)。

6.9 小结

最后一个示例再次表明，Stream API 的并行机制并不一定会提升性能。要想提升应用性能，应用必须落入到 fork/join 的领域中(对内存的 CPU 绑定处理，随机访问数据等)，而且要满足本章讨论的其他条件，包括要提供足够大的数据集以及预先元素处理等。fork/join 并行不是银弹，不过好消息是你在编写并行代码上的投资不会浪费，无论性能分析的结果是什么都是如此。你的代码会变得越来越好，当技术不断演进，从而打破了并行的成本收益平衡时，你的代码依然会有用武之地。



第7章

使用默认方法演化 API

我们可以通过默认方法为接口定义行为。为何要做这种改变，其结果又是什么呢？对第 1 个问题的简短回答就是：要支持 API 的演化。长久以来我们就一直有这个需求，不过由于要在集合中对流提供支持，这个需求才变得如此迫切(更完整的回答则是，一旦引入，默认方法还能够做其他很多事情，稍后将会看到)。Java Collections Framework(就像其扩展一样，例如 Google Guava)在很大程度上是基于接口的：也就是说，集合的功能是由接口的

Javadoc 契约定义的(也有一些例外)。要想增强现有的集合库以支持流，我们需要诸如 Collection.stream 这样的接口方法。另一个手段就是完全替换掉 Java Collections Framework，不过这会对兼容性与代码维护性造成难以察觉的影响。

因此，Java 8 的设计者们决定改变这一情况，即向已有的接口中添加方法会导致难以处理的兼容性问题。回顾这些问题有助于我们理解设计者在做出这个决策时的选择问题。例如，考虑将一个抽象方法 stream 添加到 Collection 中(注意，这里所描述的问题也适用于所有已发布的接口，而不仅仅是平台库中的那些)。在 Java 8 之前，Collection 只声明了抽象方法：

```
public interface Collection<E> {
    int size();
    boolean isEmpty();
    // 11 more abstract method declarations
}
```

一个类要想实现接口，那么它必须通过具体方法重写所有抽象方法声明：

```
class CustomCollection<E> implements Collection<E> {
    public int size() { ... }
    public boolean isEmpty() { ... }
    // 11 more concrete method declarations
}
```

现在来思考将集合作为流源这个问题。我们需要通过一个方法 stream 来扩展接口 Collection；如果没有默认方法，那么只能通过添加一个抽象方法声明来实现：

```
public interface Collection<E> {
    int size();
    boolean isEmpty();
```

```
    // 11 more abstract methods
    Stream<E> stream(); // abstract method - not added in reality
}
```

如果实现 CustomCollection 针对这个新版本的 Collection 进行编译，那么结果就是源不兼容：编译失败，因为 Collection 现在声明了一个没有被 CustomCollection 重写的抽象方法。如果没有默认方法，那么唯一一种解决方案就是添加一个具体的 stream 的重写声明，使得 CustomCollection 匹配新的 Collection 定义。不过，如果接口是库的一部分(Collection 已经得到了广泛的应用，任何公开的 API 都如此)，那就要在接口的每一次增强后要求其所有实现都必须进行扩展，这几乎是不可能的事情。

注意，如果不重新编译，那么 CustomCollection 现有的字节码还将链接到新版的 Collection 上并运行——根据 Java 语言规范(13.2 节)，接口与实现之间要做到二进制兼容。不过，不修改实现只会将问题传递给客户端程序员：现在，利用接口新功能的客户端代码将能编译通过，没有任何问题：

```
Collection<Integer> c = new CustomCollection<>();
Stream<Integer> s = c.stream();
```

不过运行时就会出现 AbstractMethodError 错误。

根本问题在于 Java 接口的传统角色被限制为定义客户端与实现之间的一种契约；提供契约所承诺功能的职责完全在于实现。自然而然地，如果扩展了契约的需求，那么提供者就要意识到这种变更，并且要满足新增加的部分。不过，当接口发布后，要让所有实现都遵循这个要求就不太现实了。因此，已发布的接口就永远不会被抽象方法所扩展。

默认方法的引入改变了这一情况，它扩展了接口的角色，除了定义契约外，接口还可以包含部分实现。现在，接口可以在两

种非抽象方法体中包含实现代码了：默认方法与静态方法。默认方法对于 API 演化来说是最为重要的；例如，下面就是 Java 8 中 Collection.stream 的声明：

```
public interface Collection<E> {
    int size();
    boolean isEmpty();
    // 11 more abstract methods
    default Stream<E> stream() {
        // delegate to spliterator, also a default method of
Collection
        return StreamSupport.stream(spliterator(), false);
    }

    // other default methods
}
```

对于客户端来说，接口的含义保持不变：它定义了一个类型，并列出客户端可以通过该类型调用的方法声明。对于实现来说，其与接口之间的关系也保持不变：实现依然需要通过具体方法重写接口中的抽象方法声明。不过，现在向接口与实现的关系中添加了一个新元素：实现可以选择重写默认方法声明(这是因为它们都是虚拟的)，替换或是改变其行为，就像传统的 Java 实例方法重写一样。

现在，我们可以扩展接口，让客户端代码调用新方法而无须对已有实现做任何修改。这扫清了 API 演化之路上的绊脚石(从 JDK 1.0 以来就存在了)。我们不禁要问，这个阻碍为何会存在如此长的时间。

做出这么重大的改变会遇到很多困难，除此之外，我们对于多继承的态度也需要转变。对多继承的厌恶之情弥漫在 Java 最初的设计过程中，这是因为多继承给其他语言带来了很多难题，特

别是 C++的“钻石问题”。即便如此，接口的多继承一直以来都为人所接受，这是因为接口本质上就是类型定义，多类型继承并不会带来相同的问题。引入默认方法会扩展接口的角色，让接口提供行为，这样现在就要允许行为的多继承了。Java 8 设计者们设计出了可以管理行为继承问题的规则；多继承的本质问题与状态的继承相关，而 Java 则永远不会支持这一点。

7.1　使用默认方法

经历过如此重大的改变后，新的最佳实践需要一段时间才会浮出水面。不过，我们已经可以在 Java 8 平台类中看到一些用例。下面是其中 4 种：

- 旨在被重写的方法：上一节指出诸如Collection.stream之类的方法是默认方法的主要用例。stream的实现会被委托给另一个默认方法spliterator(定义在Iterable中，它是Collection的父接口)。该方法的目的旨在返回一个分割迭代器，它会利用集合数据结构对内容做分区，以供不同线程处理，就像 5.4 节介绍的MappedByteBuffer分割迭代器一样。当然，其本身的默认实现无法实现该意图；对于大多数Collection类来说，它是次优的，因为它并不知晓所要分割的对象结构。我们可以通过虚方法派发让Collection实现通过具体方法(可以利用具体的特性)来重写分割迭代器。
- 一般不会被重写的方法：有些接口包含的方法声明常常会被同样的具体方法所实现。例如，Comparator.thenComparing(Comparator)返回的 Comparator 默认实现

是非常简单的：先计算 thenComparing 参数的接收者，如果返回 0，那么它就会计算所提供的 Comparator。特定实现其实是很难对此进行改进的。

与此形成了鲜明对比的示例就是 Iterator.remove。关于接口是否应该包含该方法是颇具争议的，因为很多实现都没有对其提供支持。不过，这个问题在 Java 8 中得到了缓解，因为它将 remove 变成了默认方法：

```
public interface Iterator<E> {
      default void remove() {
         throw new UnsupportedOperationException("remove");
      }
      ...
}
```

现在，我们就无须在每一个不支持 remove 的 Iterator 实现中重复这些具体的样板代码了。

- **辅助方法**：虽然函数式接口只有一个抽象方法来定义其中心目的，但对于它有多少个默认方法却没有任何限制。这样就可以添加额外的功能了。例如，我们之前已经看到使用 Predicate 及其具体变种来定义真值函数以对流进行过滤。你经常需要通过标准布尔运算符 AND、OR 与 NOT 来操作这些真值函数的结果。因此，Predicate 及其变种声明了实现这些运算符的默认方法：

```
Predicate tolkien = b ->b.getAuthors().contains("Tolkien");
Predicate lotr = b ->b.getTitle().contains("Lord");
Predicate old = b ->b.getPubDate().isBefore(Year.of(1960));
boolean firstLotrEdn = library.stream()
   .filter(tolkien.and(lotr).and(old))
   .findAny().isPresent();
```

- **便捷方法**：默认方法提供了将现有功能放到更恰当位置处的机会。例如，在 Java 8 之前，要想以反序对一个 List 进行排序，那么需要调用两个静态 Collections 方法：

```
Collections.sort(integerList,Collections.reverseOrder());
```

现在，凭借默认方法 List.sort(加上一个静态接口方法 Comparator.reverseOrder)，我们可以像下面这样以一种可读性更好的方式编写代码：

```
integerList.sort(Comparator.reverseOrder());
```

事实上，List.sort 不仅仅是便捷；可以通过高效的实现让 List 类重写 sort，其内部表示对于静态方法 Collections.sort 是不可见的。

这些接口方法的示例使得现有功能变得更加直观：针对 List 实例的 sort 方法，以及用于创建 Comparators 的工厂方法现在都被放到了这些接口中，这要比放到通用集合实用工具类 Collections 中更好。

7.2　抽象类的角色是什么

向接口添加实现功能会使得其角色更加接近于抽象类，后者也可以将抽象方法声明与实现功能放在一起。显然，我们不禁会问这样一个问题：抽象类的角色是什么呢。

在传统的 Java API(包含接口、抽象类与具体实现)中，这个问题的答案是显而易见的。例如，假设我们希望让之前章节提到的图书馆系统变得更加通用，添加电子书与有声书：它们也有标题和作者，但缺乏实体书的一些属性，如高度和页数等。重构后的域设计如下所示：

```
interface BookIntf {
    String getTitle();
    List<String> getAuthors();
    // other abstract methods
}
abstract class AbstractBook implements BookIntf {
    private String title;
    private List<String> authors;
    // other fields common to physical and ebooks
    // together with getters and setters for these fields
}
classPaperBook extends AbstractBook {
    privateint[] pageCounts;
    publicint[] getPageCounts() { return pageCounts; }
    // other fields and getter/setter methods for physical books
}
```

现在，假设我们想要在 BookIntf 上公开一个方法，它会将作者名拼接到单个字符串中并返回。这个方法就属于上一节介绍的便捷方法类型：

```
interface BookIntf {
    default String getAuthorsAsString() {
        returngetAuthors().stream().collect(joining(","));
    }
    ...
}
```

我们将其设定为默认方法，因为它完全可以根据另一个接口方法 getAuthors 定义。与之相反，定义 getAuthors 本身需要访问实例数据，这是无法在接口中声明的。不同子类共有的数据(如该示例中的 authors)依旧位于抽象类中，因此 getters 与 setters 也需要放到这里：

```
abstract class AbstractBook implements BookIntf {
```

```
    private List<String> authors;
    public String getAuthors() {
        return authors;
    }
    ...
}
```

就像该实例展示的，对实例状态的需求是我们继续使用抽象类的主要原因。默认方法还有其他一些限制，这意味着我们还是需要抽象类的：默认方法只能根据相同接口中的方法来实现(再加上任意类型中的可访问的静态方法)。此外，抽象方法可以声明与子类共享的受保护的状态与方法；这对于接口来说是无法做到的，接口中的所有声明都是公开的。

7.3　默认方法的语法

我们已经介绍了默认方法声明的形式。它们与类中具体方法声明是非常类似的。在接口中，它们与抽象方法声明是不同的，表现在其修饰符是 default，以及其体是一个块而非分号上。关于 default 关键字是否是必需的这一问题存在很多争论，因为即便不加 default，其语法也是清晰明了的——只有带有块体的接口方法声明才是默认方法。不过，强制使用 default 关键字的一个好处在于它能立刻让读者理解，这与修饰符 abstract(严格来说也不是必需的)对于抽象方法与抽象类的作用是一样的。

默认方法与具体实例方法之间主要的语法差别在于它们所允许使用的修饰符(例如 default)。显然，我们不能将默认方法声明为 abstract 或是 static 的，因为这些关键字会区别于其他接口方法(我们将在 7.5 节中介绍静态接口方法)。默认方法也不能声明为

final，因为它们必须总是要能被实例方法重写，下一节将会介绍这一点。就像所有接口方法一样，它们也需要声明为 public 的，不过这是隐式声明的；其他访问级别都是不允许的。

关键字 this 的含义与以前一样；它指的是当前对象，并且通过接口类型来引用。例如，方法 Iterator.forEachRemaining 的实现如下所示(出于演示的目的，这里作了简化)：

```
default void forEach(Consumer<T> action) {
   for (T t : this) {
      action.accept(t);
   }
}
```

关键字 super 也是可以使用的，不过只有前面加上父接口的名字才可以，下一节将会对此进行介绍。

接口中声明的默认方法不允许与 Object 的方法拥有相同的签名。例如，你不能在接口中重新定义 Object.toString。这与下一节将会介绍的默认方法继承相一致：对于这些原则来说，最为重要的是默认方法绝不能重写实例方法，无论是否是继承下来的都是如此。

7.4 默认方法与继承

我们已经看到了默认方法的使用与好处；那么其缺点呢？语言设计者在引入默认方法时所面对的挑战是要为方法继承设计一套系统，这套系统要简单且无歧义，同时还要将兼容性问题的影响降到最低(并且不能与其他特性相互影响)。要解决的最为重要的兼容性问题与方法调用解析相关，当有若干个实现可供选择，特别是这些实现包含了从父类继承下来的具体方法以及从接口中

继承下来的若干默认方法，我们该怎么办。关于方法调用解析的语言规范规则很复杂，不过这些规则是精心设计的，因此我们可以通过两个简单规则来理解它们。

第 1 个规则确保实例方法的选择要优于默认方法。这有时也称为“类优于接口”。例如，在如下代码中，FooBar 从接口 Foo 与父类 Bar 中继承了方法 hello：

```
// competing instance and default method declarations of hello()

interface Foo { default String hello() { return "Foo"; } }
class Bar { public String hello() { return "Bar"; } }

class FooBar extends Bar implements Foo {
    public static void main(String[] args) {
        System.out.print("Hello from " + new FooBar().hello());
    }
}
```

当 FooBar.main 运行时，输出是 Hello from Bar。无论实例方法是声明在类中还是从父类中继承的，无论实例方法是抽象还是具体，这个原则都有效。这个规则背后的动机是要防止行为性不兼容：也就是说，增加了默认方法会导致实现类的行为发生变化。如果类的优先级总是最高，那么一个类调用从父类中继承的方法时就会继续调用该方法，即便该类所实现的接口引入了它自己的与之匹配的方法声明亦如此。

第 2 个原则确保如果类继承了多个相同的默认方法，那么没有重写的默认方法会被选择。“没有重写的方法”(这不是个标准术语)指的是默认方法没有被该类继承的其他接口所重写。例如：

```
    // competing default method declarations of hello():
    // Bar.hello() overrides Foo.hello(), so is a non-overridden
method
```

```
interface Foo { default String hello() { return "Foo"; } }
interface Bar extends Foo { default String hello() { return
"Bar"; } }

class FooBar implements Foo, Bar {
    public static void main(String[] args) {
        System.out.print("Hello from " + new FooBar().hello());
    }
}
```

当 FooBar.main 运行时，输出是 Hello from Bar。

当然，如果类所继承的没有被重写的默认方法数多于 1 个，那就没有单独的没有被重写的默认方法。例如：

```
// competing default method declarations of hello():
// neither overrides the other

interface Foo { default String hello() { return "Foo"; } }
interface Bar { default String hello() { return "Bar"; } }

class FooBar implements Foo, Bar {
    public static void main(String[] args) {
        System.out.print(new FooBar().hello());
    }
}
```

在该示例中，FooBar 将会编译失败，错误消息如下所示：

```
Error: class FooBar inherits unrelated defaults for hello() from
types Foo and Bar
```

这个结果是合情合理的；编译器不知道该从两个继承的方法中选择哪一个，因此提示你消除调用歧义。可以让 FooBar 本身重写 hello，通过下面这种语法形式(Java 已经提供了，不过到目前为止只用在内部类中，目的与此还不同)来选择其中一个方法：

```
class FooBar implements Foo, Bar {
    public String hello() {
      return Bar.super.hello();
    }
}
```

相同的语法也可以用在接口默认方法体中。注意，它只能用于解析冲突，而不是重写这两个主要原则。因此，不能通过它选择一个并不是非重写的方法：

```
    // Foo.hello() is overridden by Bar.hello(), which is inherited
by FooBar

    interface Foo { default String hello() { return "Foo"; } }
    interface Bar extends Foo { default String hello() { return
"Bar"; } }

    class FooBar implements Bar {
        public String hello() {
            return Foo.super.hello();  // illegal
        }
    }
```

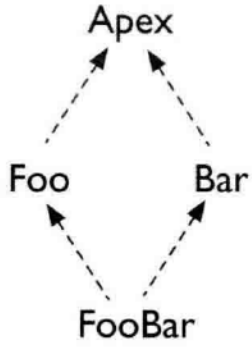
即便 FooBar 直接实现了 Foo 与 Bar，Foo.hello 也不是非重写方法，因此无法通过 super 语法选择。

这些原则是如何帮助解决“钻石问题”的呢？如下代码示例展示了一个类通过两个不同路径继承了一个声明：

```
interface Apex { default String hello() { return "Apex"; } }
interface Foo extends Apex {}
interface Bar extends Apex {}

class FooBar implements Foo, Bar {
    public static void main(String[] args) {
        System.out.print(new FooBar().hello());
    }
}
```

这个问题名字的由来与其类图的形状息息相关：



本节介绍的原则对这种场景及其变种提供了直观的解释说明：在上述代码中，FooBar 所继承的 hello 实现是由 Apex 定义的；没有其他可能性。不过，如果现在 Foo 或是 Bar 也声明了 hello 的默认实现，那么根据“非重写方法”原则，该方法就会被 FooBar 继承。注意，该原则加上虚方法分派在该情况下会得到令人惊奇的结果。例如，如果将 Foo 的上述声明改成下面这样：

```
interface Foo extends Apex { default String hello() { return
"Foo"; } }
```

那么如下代码：

```
Bar bar = new FooBar();
System.out.println("Hello from " + bar.hello());
```

就会得到输出 Hello from Foo。bar 的静态类型并不重要，重要的是它指向的是 FooBar 的实例，它的非重写方法继承自 Foo。

如果 Foo 与 Bar 都声明了默认实现，那么它们就冲突了，FooBar 必须要提供一个重写的声明才行。

兼容性问题

上一节给出的原则涵盖了绝大多数的实际情况。但是，我们

无法总是能够避免不兼容的问题。本节将会介绍 3 种会产生问题的情况。

在如下代码中，Intf 表示的是一个已发布的接口，Impl 表示的是库实现。它们不一定位于同一个库中，因此 Intf 的维护者可能对其实现一无所知，也无法在演化 API 时考虑其声明。

```
interface Intf {
    // various abstract and default methods
}
class Impl implements Intf {
    // implementation of various abstract methods in Intf
    public String hello(long l) { return "Foo"; }
}
class Client {
    public static void main(String[] args) {
        System.out.println("Hello from " + new Impl().hello(3));
    }
}
```

对于这个版本的代码来说，运行 Client.main 显然会产生输出 Hello from Impl。现在考虑对 Intf 做两个不同的变更，每个变更都会向 Intf 添加一个方法，并且该方法几乎与 Impl.hello 的声明相匹配。例如，Intf 会声明一个 hello，其签名相同，不过有一个不兼容的返回值：

```
interface Intf {
    default void hello(long l) { }
    ...
}
```

如果是重载方法，那么类中的总是占优；不过这并非重载方法。编译 Impl 与新版的 Intf 会生成如下消息：

```
Error: foo(long) in Impl cannot implement foo(long) in Intf
```

```
return type java.lang.String is not compatible with void
```

如果 Impl 的 hello 声明不是公共的，那么该问题就延伸了；在这种情况下，即便 Intf 中新方法的签名是精确匹配的，但也会导致编译器报错，说 Impl 的重写会缩小 hello 的访问范围(因为接口方法自动就是公共的)。

第 2 个问题更加严重。接口添加了一个默认方法声明，该方法带有一个参数，其类型与现有声明中的相应参数是可赋值兼容的。对于该示例来说，新的声明可能如下所示：

```
interface Intf {
   default String hello(int l) { return "Intf"; }
   ...
}
```

根据方法重载的原则，上述代码会声明一个新的 hello 重载方法，由于它没有重写已有方法，因此会被 Impl 继承。这样，新老方法重载对于 Client 发出的 hello(3)调用都是适用的，当编译 Client 时，最具体的方法会被选择，这是 Java 语言规范(15.12 节)定义的。由于 3 是 int 而非 long 类型，因此新继承的重载方法就更加具体，输出将变为 Hello from Intf。Impl 没有做任何修改，但其行为却发生了变化！该示例说明对于已有的复杂语义规则来说，要想实现完全兼容的语言该是多么困难(从这个角度来说，方法重载是非常困难的事情，这在 2.8 节中已经介绍过了)。

另一种行为型不兼容与语法问题无关，而是动态方法分派本身所固有的。新引入的父类型方法可能无法保证实现类的不变性，因为它对实现类一无所知。下面来看实际例子，即 Java 8 引入的 Map.putIfAbsent：

```
default V putIfAbsent(K key, V value) {
   V v = get(key);
```

```
    if (v == null) {
        v = put(key, value);
    }
    return v;
}
```

如果没有重写，那么该方法就会破坏任何实现的 Map 的线程安全性：在线程进行测试与执行动作之间，v 值可能会被另一个线程设置。当前线程则会重写该值，这与该方法的规范相反。这个问题没有真正的解决方案；在最坏的情况下，我们要检查所有实现，从而确保它们重写了新引入的默认方法。

注意，本节所介绍的问题并不是什么新问题：它们都可能出现在类继承中。类继承关系的变化会导致这类问题的发生，更为严重的是，接口变化会产生这些影响对于 Java 来说是全新的事情。幸好，在实际情况下，这些问题并不会经常出现。

7.5　接口中的静态方法

一旦做了可以在接口中提供行为的决定后，很自然地我们会想到静态方法，看看它是否可以，或是应该允许出现在接口中。我们很容易就能看到允许这么做的好处：在整本书中，我们都会看到已经在这么做了，如下代码所示：

```
Stream.of(1,2,3);
Comparator.naturalOrder();
```

不过，从本章之前章节的视角来看，我们已经认识到了减少兼容性问题的重要性。实际上，对于静态接口方法来说，这些问题是可以消除的，方式是将用于引用它们的语法限制为 DeclaringInterface.MethodName 这种形式(显然，该解决方案对于

默认方法是无效的，因为它与虚方法分派不兼容)。因此，与静态类方法的一个差别就是静态接口方法不能继承：

```
interface Bar {static void barHello() {} }
class Foo { static void fooHello() {} }

class FooBar extends Foo implements Bar {
   public static void main(String[] args) {
      fooHello();     // invoke inherited method
      barHello();     // illegal - doesn't compile
      Bar.barHello(); // only legal way to reference barHello
   }
}
```

出于同样的原因，你不能通过语法ObjectReference.MethodName引用静态接口方法[1]。静态接口方法在其他方面则是以相同方式声明的，并且与静态类方法拥有几乎相同的属性。就像其他接口方法一样，可以将其声明为public，不过其隐式访问级别也是public。不允许将静态接口方法声明为final，因为这对于无法被继承的方法来说是个毫无意义的修饰符。

使用静态方法

现在，想知道 API 设计中会出现哪些惯用法来使用静态接口方法还为时尚早。在 Java 8 中，平台类将其公开为工厂方法(例如 Stream.of、Collector.of 以及 Comparator.comparing)，并且作为对传统实用工具类的一种改进方式。

实用工具类在定位功能上存在很多困难。平台库 java.util.Collections 就是最为极端的例子，它包含了 60 多个静态

[1] 与静态类方法的这两个差别(不允许继承，不允许通过对象引用调用)可以看作是防止出现问题的两个手段。

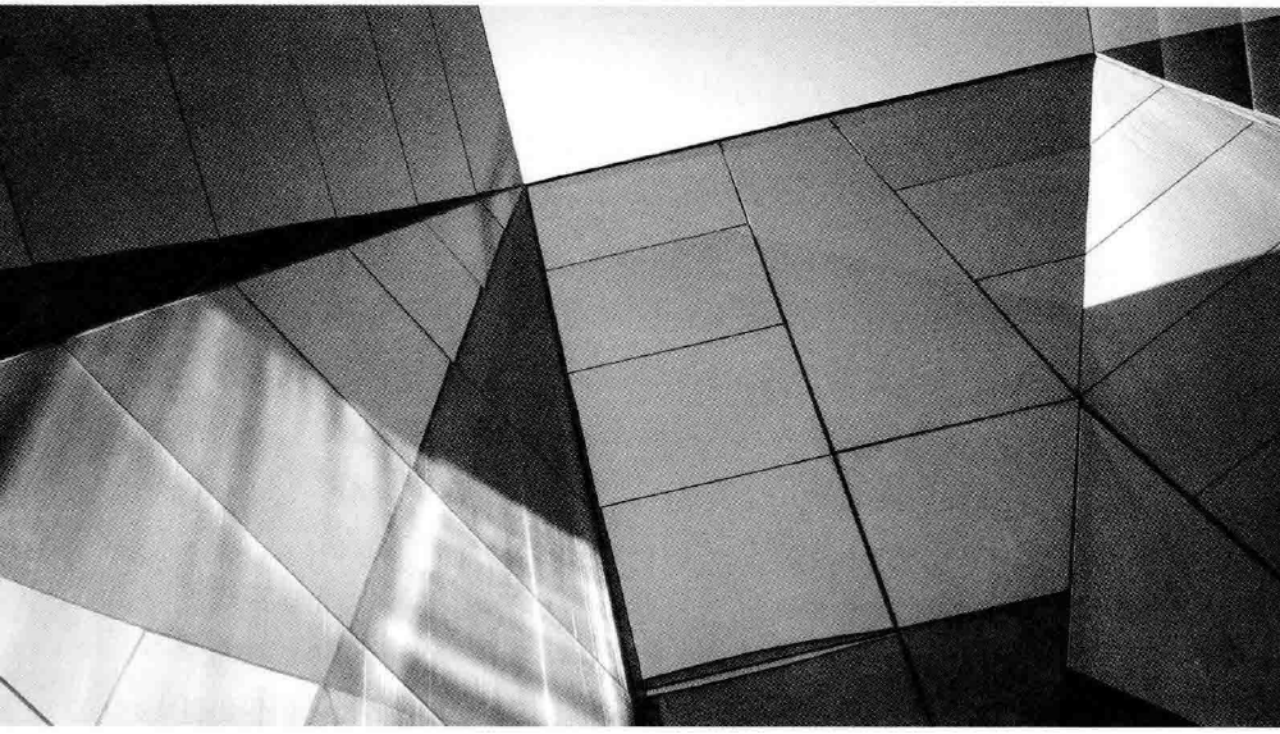
方法，其中大多数都与通用集合没什么关系，而是与 Set、List、Queue 及 Map 有关(Map 甚至不是 Collection 的子类型)。Java Collections Framework 的新手无法以系统化方式推断出到底该去哪里寻找用于生成 Map 实例的工厂方法。这种功能的随意分配是学习 API 的一个主要障碍。与之相反，既然接口可以公开诸如 Comparator.comparing 与 Stream.of 这样的方法，那么开发者自然而然就期望到里面去寻找所需的方法了。这种改进并不限于新方法；例如，Java 8 引入了名为 Comparator.reverseOrder 的工厂方法，其实现只不过是被委托给了已有的方法 Collections.reverseOrder 而已(不过这个方法很难找到)。

即便如此，诸如 Collections 之类的实用工具类也不会消失在新 APIs 中。考虑接口 Collector；在 Stream API 中有 40 多个针对该接口的工厂方法。其中只有两个被声明为接口本身的静态方法，也就是 4.3 节介绍的两个通用的 Collector.of 的重载方法。4.1 节介绍的 40 多个奇怪的预定义工厂方法都继续存在，并且还有了自己的类 Collectors；将它们放到 Collector 接口中会淹没其核心功能。通过这个示例我们认识到，静态接口方法与实用工具类之间的平衡随着时间的推移终将会实现。

7.6　小结

想改变一门已经广泛使用了近 20 年的编程语言并不是那么轻松的事情。如果想要修改的是语言的核心特性，就像接口之于 Java，那么问题就会变得更加困难。虽然本章谈到了很多问题，但向 Java 接口添加行为这个举动从总体上来看并没有造成太大的困难。

从一定程度上来说，这是因为特性已经根据促进 API 演化这个目的而得到了严格的裁剪。围绕着引入默认方法的一些争论基于这个想法：这会将其他语言中的特质(Trait)或是混入(Mixin)等特性加入进来。不过，由于接口中没有状态，因此这个想法难以实现。与之类似，就像本章所介绍的那样，说接口可以替换掉抽象类，或是说实用工具类是多余的等想法都是言过其实的。不过，它们的确实现了设计的目标，促进了 API 的演化；Stream API 就是最初的受益者；同时，这还使得其他 API 的可维护性变得更好，功能也得到了增强。



本书总结

在本书行将结束之际我想谈谈个人看法。在过去40多年间，我已经记不清有多少技术曾宣称自己将成为业界的未来，不过其中很多技术在一两年后就渐渐淡出人们的视线。有极少量的技术一直在慢慢地发展着：这些技术拥有坚定的拥护者，但却一直没有成为商业主流。在这其中引人注目的就是函数式编程，它吸引着最聪明的一群人，也产生了软件开发中最棒的理念，但同时却只是小众人群喜爱的东西。

从我个人角度来说，虽然早在20世纪80年代我就开始函数式编程了，但我却并没有成为一名函数式程序员。相反，我跟随

着业界主流，迈向了面向对象编程的行列，最后选择了 Java。将近 20 年后，从大多数角度来看，Java 已经成为了这个世界上最为流行的编程语言了。不过在过去几年间，我并没有对自己的选择感到得意；只知道 Java 的程序员错过了其他竞争语言中出现的很多好用的新编程技术。很多新技术(如延迟计算、闭包与模式匹配等)都源自于函数式语言。这种趋势依旧在持续着；函数式程序员对自己的未来持乐观态度，这主要是因为硬件制造技术与成本方面的趋势表明大规模的并发系统将会在未来占据统治地位。数据不变性将成为推出这种系统的关键所在。

Java 并不会成为一门函数式语言，不过 Java 程序员应该能够享受到函数式语言所带来的一些理念。Java 8 的变化就是向这个方向前进的第一步。这些变化向实际的 Java 编程引入了不变性与延迟计算，在一定程度上解决了将任务分派到多个处理器上执行的巨大挑战。虽然承诺向后兼容使得对一门拥有 20 年历史的语言的任何改变都变得非常困难，但 Java 设计团队展现出了过人的能力，他们将函数式编程的理念集成到了拥有完全不同设计原则的语言中。他们取得了巨大的成功，这是 Java 历史上最大的一组变化。

我很激动能看到这些变化，并且希望将这些激动人心的事情传递给你。我希望这本书能够帮助你理解 Java 的新方向，从长远来看能够加入到语言的未来中去。现在我想说的是：编程应该总是让人享受的，不过编写 Java 8 支持的简洁、可读性好，并且高效的代码时会让你更加享受其中。我很高兴这本书能对你的编程生涯起到这种变化作用。

Maurice Naftalin

lambda表达式权威指南

《精通lambda表达式：Java多核编程》介绍Java SE 8中与lambda相关的特性是如何帮助Java迎接下一代并行硬件架构的挑战的。本书讲解了如何编写lambda、如何在流与集合处理中使用lambda，并且提供了完整的代码示例。你将学习如何通过lambda表达式充分利用当今多核硬件所带来的性能改进。

主要内容：

- 为何需要lambda，它将如何改变Java编程
- lambda表达式语法
- 流与管道的基本操作
- 使用收集器与汇聚来终止管道
- 创建流
- 分割迭代器、fork/join框架与异常
- 使用微基准测试检查流的性能
- 使用默认方法演化API

本书代码示例可以从OraclePressBooks.com网站下载

Oracle Press

ISBN 978-7-302-40553-5

9 787302 405535

定价：39.00元

清华大学出版社数字出版网站

WQBook 书文局泉

www.wqbook.com

McGraw-Hill

全球智慧中文化

http://www.mheducation.com