原文：<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html>

翻译：<http://zh.lucida.me/blog/java-8-lambdas-inside-out-library-features/>

# 关于

Java SE 8增加了新的语言特性（例如lambda表达式和默认方法），为了Java SE 8的类库也进行了很多改进，本文简要介绍了这些改进。

# 背景

自从lambda表达式称为Java语言的一部分之后，Java集合API就面临着大幅变化。而JSR 355（规定了Java lambda表达式的标准）的正式启用更是使得Java集合API变得过时不堪。尽管可以从头实现一个新的集合框架（比如"Collection II"），但取代现有的集合框架是一项非常艰难的工作，因为集合接口渗透了Java生态系统的每个角落，将它们一一替换为新类库需要相当长的时间。因此，Java SE 8采取了演化的策略（而非推倒重来）以改进集合API：

* 为现有的接口（例如Collection、List和Stream）添加扩展方法
* 在类库中增加新的流（stream，即java.util.stream.Stream）抽象以便进行聚合操作
* 改造现有的类型使之可以提供流视图（stream view）
* 改造现有的类型使之可以容易的使用新的编程模式，这样用户就不必抛弃使用已久的类库，例如ArrayList和HashMap（当然，这并不是说集合API会长驻永存，毕竟集合API在设计之初并没有考虑到lambda表达式，未来可能会在JDK中添加一个更现代的集合类库）

除了上面的改进，还有一项重要工作就是提供更加易用的并行库。尽管Java平台已经对并行和并发提供了强有力的支持，然而开发者在实际工作（将串行代码并行化）中仍然会碰到很多问题，因此，使得Java类库对串行和并行都友好是很重要的。将重点从具体执行细节（how computation should be formed）转移到抽象执行步骤（what computation should be performed）对实现这一点很有帮助。另外，平衡并行的易用性和可见性也是很重要的一点，Java并行库的目标是提供显式但非侵入的并行。（如果把并行变的透明，很可能会引入不确定性以及各种数据竞争问题）。

# 内部迭代vs外部迭代

集合类库主要依赖于外部迭代。Collection实现了Iterator接口，使得用户可以依次遍历集合中的元素，比如我们需要把一个形状集合中的所有形状颜色都设置为红色，可以这么写：

|  |
| --- |
| for (Shape shape : shapes) {  shape.setColor(RED);  } |

以上例子演示了外部迭代：for-each循环调用shapes的iterator()方法进行依次遍历。外部循环的代码非常直接，但它有如下问题：

* Java的for循环是串行的，而且必须按照集合中元素的顺序进行依次处理
* 集合框架无法对控制流进行优化，例如通过排序、并行、短路求值以及惰性求值改善性能

尽管有时for-each循环的这些特性（串行、依次）是我们所期待的，但它对改善性能造成了阻碍。

我们可以使用内部迭代替代外部迭代，用户把对迭代的控制权交给类库，并向类库传递迭代时所需执行的代码。下面是前例的内部迭代代码：

|  |
| --- |
| shapes.forEach(s -> s.setColor(RED)); |

尽管看起来是只是一个小小的语法改动，但是它们的实际差别非常巨大。用户把对操作的控制权交给类库，从而允许类库进行各种各样的优化（例如乱序执行、惰性求值、并行等）。总的来说，内部迭代使得外部迭代中不可能实现的优化成为可能。

外部迭代同时承担了做什么（把形状设置为红色）和怎么做（得到Iterator实例然后依次遍历）两项职责，而内部迭代只负责做什么，而把怎么做留给类库。通过这样的职责转变：用户的代码会变得更加清晰，而类库则可以进行各种优化，从而使所有用户从中受益。

# 流

流是Java SE 8类库中新增的关键抽象，定义在java.util.stream包中（这个包里有若干流类型：Stream<T>代表对象引用流，此外还有一系列特化流，比如IntStream代表整型数字流）。每个流代表一个值序列，流提供一些列常用的聚合操作，使得我们可以在流上方便的进行各种运算。另外，集合类库也提供了一些便捷的方式，使得我们可以用操作流的方式操作集合、数组以及其他数据结构。

流的操作可以被组合成流水线，以前面的例子为例，如果我们只想把蓝色改成红色：

|  |
| --- |
| shapes.stream()  .filter(s -> s.getColor() == BLUE)  .forEach(s -> s.setColor(RED)); |

在Collections上调用stream(0会生成该集合元素的流视图，接下来，filter()操作会产生只包含蓝色形状的流，最后，这些蓝色形状会被forEach操作设置为红色。

如果我们想把蓝色形状提取到新的List里，则可以：

|  |
| --- |
| List<Shape> blue =  shapes.stream()  .filter(s -> s.getColor() == BLUE)  .collect(Collectors.toList()); |

collect()操作会把接收到的元素聚合在一起，其方法参数则指定如何进行聚合操作。这里，我们使用toList()把元素输出到List中。

如果每个形状都被保存在Box里，假设我们想要知道哪个盒子至少包含一个蓝色形状，则可以这么写：

|  |
| --- |
| Set<Box> hasBlueShape =  shapes.stream()  .filter(s -> s.getColor() == BLUE)  .map(s -> s.getContainingBox())  .collect(Collectors.toSet()); |

map()操作通过映射函数（这里的映射函数接收一个形状，返回包含它的盒子）对输入流里的元素进行依次转换，然后产生新流，最后通过collect()把元素输出到Set中。

如果我们想要得到蓝色形状的总重量，可以这么写：

|  |
| --- |
| int sum =  shapes.stream()  .filter(s -> s.getColor() == BLUE)  .mapToInt(s -> s.getWeight())  .sum(); |

以上这些例子演示了流框架的设计，以及如何使用流框架解决实际问题。

# 流vs集合

集合和流尽管在表面上看起来很相似，但它们的设计目标是不同的：集合主要用来对元素进行有效管理和访问，而流并不支持对元素进行直接操作或直接访问，只支持通过声明式操作对元素进行计算并得到结果。除此之外，流和集合还有一些其他不同：

* 无存储：流并不存储值。流的元素源自数据源（可能是某个数据结构、生成函数或I/O通道等），并通过一系列的计算步骤得到
* 天然的函数式风格：对流的操作会产生一个结果，但不会修改对应的数据源
* 惰性求值：多数流操作（包括过滤、映射、排序以及去重）都可以以惰性方式实现，这使得我们可以用一遍遍历完整整个流水线操作，并可以用短路操作提供更高效的实现
* 无需上界：不少问题都可以被表达为无效流，即用户不停地读取流，直到满意的结果出现为止。集合是有限的，但流不是（操作无限流时必须使用短路操作，以确保操作可以在有限时间内完成）

从API的角度来说，流和集合完全相互独立，不过我们可以把集合作为流的数据源（Collection拥有stream()和parallelStream()方法），也可以通过流产生一个集合（使用collect()方法）。Collection以外的类型也可以作为流的数据源，比如Jdk中的BufferedReader、Random和BitSet已经被改造为可以用作流的数据源，Arrays.stream()则产生给定数组的流视图。事实上，任何可以用Iterator描述的对象都可以作为流的数据源，并且如果有额外的信息（比如大小、是否有序等），库还可以进行进一步的优化。

# 惰性

过滤和映射这样的操作既可以被急性求值（以filter为例，急性求值需要在方法返回前完成对所有元素的过滤），也可以被惰性求值（用Stream代表过滤结果，当且仅当需要时才进行过滤操作）。在实际中进行惰性计算可以带来很多好处，比如说，如果我们进行惰性过滤，就可以把过滤和流水线里的其他操作混合在一起，这样就不需要对数据进行多次遍历。类似的，如果要在一个大型集合中搜索第一个满足某个条件的元素，我们可以在找到后直接停止，而不是继续处理整个集合。（这一点对于无限数据源来说尤其重要。惰性求值对于有限数据源来说仅仅是一个优化，但它使得对无限数据源的操作成为可能，没有惰性求值，对无限数据源的操作将无法终止）。

对于过滤和映射这样的操作，不管它们是如何实现的，我们很自然的会把它们当做是惰性求值操作，而对于sum()这样的生成值的操作或者forEach()这样产生副作用的操作都是天然的急切求值操作，因为它们必须要产生具体的结果。

以下面的流水线为例：

|  |
| --- |
| int sum =  shapes.stream()  .filter(s -> s.getColor() == BLUE)  .mapToInt(s -> s.getWeight())  .sum(); |

这里的过滤操作和映射操作都是惰性的，这意味着在调用sum()之前，不会从数据源提取任何元素。在执行sum操作时，会把过滤、映射以及求和混合在对数据源的一遍遍历中，这样可以大大减少维持中间结果所带来的开销。

大多数循环都可以用数据源（数组、集合、生产函数以及I/O管道）上的聚合操作来表示：进行一系列惰性操作（过滤和映射等操作），然后用一个急性求值操作（forEach、toArray、collect等操作）得到最终结果，比如过滤-映射-累积、过滤-映射-排序-遍历等组合操作。惰性操作一般被用来计算中间结果，这一点在Streams API中得到了良好的体现：与其让filter和map返回一个集合，倒不如让它们返回一个新的流。在Streams API中，返回流对象的操作都是惰性操作，而返回非流对象的操作（或者无返回值的操作，例如forEach()）都是急性操作。绝大多数情况下，惰性操作会被用于聚合，这正是我们想要的：流水线中，每个阶段都会接收输入流中的元素，进行转换，并把转换结果传递给下一个阶段。

在使用这种“数据源-惰性操作-惰性操作-急性操作”流水线时，流水线中的惰性几乎是不可见的，因为计算过程被夹在数据源和最终结果（或副作用操作）之间，这使得API的可用性和性能得到了改善。

对于anyMatch(Predicate)和findFirst()这些急性求值操作，我们可以使用短路来终止不必要的运算，以下面的流水线为例：

|  |
| --- |
| Optional<Shape> firstBlue =  shapes.stream()  .filter(s -> s.getColor() == BLUE)  .findFirst(); |

由于过滤这一步是惰性的，findFirst(0在从其上游得到一个元素后就会终止，这意味着我们只会处理这个元素及其之前的元素，而不是所有元素。

在这种设计下，用户并不需要显式进行惰性求值，甚至都不需要了解惰性求值，类库会自行选择最优化的计算方式。

# 并行

流水线既可以串行执行，也可以并行执行，并行或串行是流的属性。除非你显式要求使用并行流，否则Jdk总会返回串行流。（串行流可以通过parallel()方法被转换为并行流）。

尽管并行是显式的，但它并不需要成为侵入式的，利用parallelStream()，我们可以轻松的把之前重量求和的代码并行化：

|  |
| --- |
| int sum =  shapes.parallelStream()  .filter(s -> s.getColor = BLUE)  .mapToInt(s -> s.getWeight())  .sum(); |

并行化之后和之前的代码区别并不大，但我们可以很容易看出它是并行的，并且我们不需要自己去实现并行代码。

因为流的数据源可能是一个可变集合，如果在遍历流时，数据源被修改，就会产生干扰。所以，在进行流操作时，流的数据源应保持不变。这个条件并不难维持，如果集合只属于当前线程，只要lambda表达式不修改流的数据源就行。（这个条件和遍历集合时所需的条件相似，如果集合在遍历时被修改，绝大多数的集合实现都会抛出ConcurrentModificationException）。这个需求被称为无干扰性。

我们应避免在传递给流方法的lambda表达式中产生副作用。一般来说，打印调试语句这种输出变量的操作是安全的，但是，在lambda表达式中访问可变变量就有可能造成数据竞争或者其他意想不到的问题，因为lambda在执行时可能会同时运行在多个线程上，因此它们看到的元素有可能和正常的顺序不一致。

无干扰性有两层含义：

1、不要干扰数据源

2、不要干扰其他lambda表达式，当一个lambda修改某个可变状态，而另一个lambda读取该状态时，就会产生干扰

只要满足无干扰性，我们就可以安全的进行并行操作并得到可预测的结果，即便对线程不安全的集合（例如ArrayList）也是一样。

# 实例

下面的代码源于Jdk中的Class类型（getEnclosingMethod方法），这段代码会遍历所有声明的方法，然后根据方法名称、返回类型以及参数的数量和类型进行匹配：

|  |
| --- |
| for (Method method : enclosingInfo.getEnclosingClass().getDeclaredMethods()) {  if (method.getName().equals(enclosingInfo.getName())) {  Class<?>[] candidateParamClasses = method.getParameterTypes();  if (candidateParamClasses.length == parameterClasses.length) {  boolean matches = true;  for (int i = 0; i < candidateParamClasses.length; i += 1) {  if (!candidateParamClasses[i].equals(parameterClasses[i])) {  matches = false;  break;  }  }  if (matches) { // finally, check return type  if (method.getReturnType().equals(returnType)) {  return method;  }  }  }  }  }  throw new InternalError("Enclosing method not found"); |

通过使用流，我们不但可以消除上面代码里所有的临时碧娜来那个，还可以把控制逻辑交给类库处理。通过反射得到方法列表之后，我们利用Arrays.stream()将其转化为Stream，然后利用一系列过滤器去除类型不符、参数不符以及返回值不符的方法，接着通过findFirst得到Optional<Method>，最后利用orElseThrow返回返回值或者抛出异常。

|  |
| --- |
| return Arrays.stream(enclosingInfo.getEnclosingClass().getDeclaredMethods())  .filter(m -> Objects.equals(m.getName(), enclosingInfo.getName()))  .filter(m -> Arrays.equals(m.getParameterTypes(), parameterClasses))  .filter(m -> Objects.equals(m.getReturnType(), returnType))  .findFirst()  .orElseThrow(() -> new InternalError("Enclosing method not found")); |

|  |
| --- |
|  |

相对于未使用流的代码，这段代码更加紧凑，可读性更好，也不容易出错。

流操作特别适合对集合进行查询操作。假设有一个“音乐库”应用，每个库都有一个专辑列表，每张专辑列表都有其名称和音轨列表，每首音轨表都有名称、艺术家和评分。假设我们需要得到一个按名字排序的专辑列表，专辑列表里的每张专辑都至少包含一首四星及四星以上的音轨。为了构建这个专辑列表，我们可以这么写：

|  |
| --- |
| List<Album> favs = new ArrayList<>();  for (Album album : albums) {  boolean hasFavorite = false;  for (Track track : album.tracks) {  if (track.rating >= 4) {  hasFavorite = true;  break;  }  }  if (hasFavorite)  favs.add(album);  }  Collections.sort(favs, new Comparator<Album>() {  public int compare(Album a1, Album a2) {  return a1.name.compareTo(a2.name);  }  }); |

我们可以用流操作来完成上面代码中的三个主要步骤：识别一张专辑中是否包含至少一首四星及以上的音轨；按名字排序；把满足条件的专辑放到一个List中。

|  |
| --- |
| List<Album> sortedFavs =  albums.stream()  .filter(a -> a.tracks.anyMatch(t -> (t.rating >= 4)))  .sorted(Comparator.comparing(a -> a.name))  .collect(Collectors.toList()); |

# 收集器

在之前的例子汇总，我们利用collect()方法把流汇总的元素聚合到List或Set中。collect()接收一个类型为Collector的参数，该参数决定了如何把流中的元素聚合到其他数据结构中。Collectors类中包含了大量常用收集器的工厂方法，toList()和toSet()就是其中最常见的两个。

Collector的类型由其输入类型和输出类型决定。以toList()收集器为例，它的输入类型为T，输出类型为List<T>，toMap是另外一个较为复杂的Collector，它有若干个版本。最简单的版本接收一对函数作为输入，其中一个函数用来生成键（key），另一个函数用来生成值（value）。toMap的输入类型是T，输出类型是Map<K,V>，其中K和V分别是前面两个函数所生成的键类型和值类型。（复杂版本的toMap收集器则允许你指定目标Map的类型或解决键冲突）。举例来说，下面的代码以目录数字为键值创建一个倒排索引：

|  |
| --- |
| Map<Integer, Album> albumsByCatalogNumber =  albums.stream()  .collect(Collectors.toMap(a -> a.getCatalogNumber(), a -> a)); |

groupingBy是一个与toMap相类似的收集器，比如说我们想要把我们最喜欢的音乐按歌手列出来，这时我们就需要这样的Collector：它以Track作为输入，以Map<Artist,List<Track>>作为输出。groupingBy收集器就可以胜任这个工作，它接收分类函数，然后根据这个函数生成Map，该Map的键是分类函数的返回结果，值是该分类下的元素列表。

|  |
| --- |
| Map<Artist, List<Track>> favsByArtist =  tracks.stream()  .filter(t -> t.rating >= 4)  .collect(Collectors.groupingBy(t -> t.artist)); |

收集器可以通过组合和复用来生成更加复杂的收集器，简单版本的groupingBy收集器根据分类函数（这里是音轨的艺术家）将元素组织到桶里，并将映射到同一个桶里的所有元素放到一个列表中。除了这个版本，还有一个更加通用的版本允许你使用其它收集器来整理输入元素：它接收一个分类函数以及一个下游收集器（单参数版本的groupingBy使用toList()作为其默认下游收集器）。举例来说，如果我们想把每首歌曲的演唱者收集到Set而非List中，我们可以使用toSet收集器：

|  |
| --- |
| Map<Artist, Set<Track>> favsByArtist =  tracks.stream()  .filter(t -> t.rating >= 4)  .collect(Collectors.groupingBy(t -> t.artist,  Collectors.toSet())); |

如果我们需要按照歌手和评分来管理歌曲，我们可以生成多级Map：

|  |
| --- |
| Map<Artist, Map<Integer, List<Track>>> byArtistAndRating =  tracks.stream()  .collect(groupingBy(t -> t.artist,  groupingBy(t -> t.rating))); |

在最后的例子里，我们创建了一个歌曲标题里面的词频分布。我们首先使用Stream.flatMap()得到一个歌曲流，然后用Pattern.splitAsStream把每首歌曲的标题打散成词流；接下来我们用groupingBy和String.toUpperCase对这些词进行不区分大小写的分组，最后使用counting()收集器计算每个词出现的次数（从而无需创建中间集合）。

|  |
| --- |
| Pattern pattern = Pattern.compile("\\s+");  Map<String, Integer> wordFreq =  tracks.stream()  .flatMap(t -> pattern.splitAsStream(t.name)) // Stream<String>  .collect(groupingBy(s -> s.toUpperCase(), counting())); |

flatMap接收一个返回流（这里是歌曲标题里的词）的函数，它利用这个函数将输入流中的每个元素转换为对应的流，然后把这些流拼接到一个流中。所以上面代码中的flatMap会返回所有歌曲标题里面的词，接下来我们不区分大小写的把这些词分组，并把词频作为值（value）储存。

Collectors类包含大量的方法，这些方法被用来创造各式各样的收集器，以便进行查询、列表（tabulation）和分组等工作，当然你也可以实现一个自定义Collector。

# 并行的实质

Java SE 7引入了Fork/Join模型，以便高效实现并行计算。不过，通过Fork/Join编写的并行代码和同功能的串行代码的差别非常巨大，这使改写串行代码变的非常困难。通过提供串行流和并行流，用户可以在串行操作和并行操作之间进行便捷的切换（无需重写代码），从而使得编写正确的并行代码变的更加容易。

为了实现并行计算，我们一般要把计算过程递归分解（recursivedecompose）为若干步：

* 把问题分解为子问题
* 串行解决子问题从而得到部分结果（partialresult）
* 合并部分结果合为最终结果

这也是Fork/Join的实现原理。

为了能够并行化任意流上的所有操作，我们把流抽象为Spliterator，Spliterator是对传统迭代器概念的一个泛化。分割迭代器（spliterator）既支持顺序依次访问数据，也支持分解数据：就像Iterator允许你跳过一个元素然后保留剩下的元素，Spliterator允许你把输入元素的一部分（一般来说是一半）切开放到另一个新的Spliterator中，而剩下的数据则会被保存在原来的Spliterator里。（这两个分割迭代器还可以被进一步分解。）除此之外，分割迭代器还可以提供源的元数据（比如元素的数量，如果已知的话）和其它一系列布尔值特征（比如说“元素是否被排序”这样的特征），Streams框架可以利用这些数据来优化执行。

大多数用户无需去实现Spliterator接口，集合上的stream()方法往往就足够了。但如果需要实现一个集合或一个流，那可能需要手动实现Spliterator接口。Spliterator接口的API如下所示：

|  |
| --- |
| public interface Spliterator<T> {  // Element access  boolean tryAdvance(Consumer< ? super T> action);  void forEachRemaining(Consumer< ? super T> action);  // Decomposition  Spliterator<T> trySplit();  //Optional metadata  long estimateSize();  int characteristics();  Comparator< ? super T> getComparator();  } |

集合库中的基础接口Collection和Iterable都实现了正确但相对低效的spliterator()实现，但子接口（例如Set）和具体实现类（例如ArrayList）均提供了高效的分割迭代器实现。分割迭代器的实现质量会影响到流操作的执行效率，从split()方法中返回平衡的划分将提高CPU利用率，另外，提供正确的特征和大小等元数据也有利于进一步优化。

# Jdk中的流和lambda

Stream在Java SE 8中非常重要，Jdk中广泛的使用了Stream，比如Collection提供了stream()和parallelStream()把集合转化为流，数组可以通过Arrays.stream()被转化为流。除此之外，Stream中还有一些静态工厂方法（以及相关的原始类型流实现）被用来创建流，例如Stream.of()、Stream.generate()以及IntStream.range()。其它的许多类也提供了流相关的方法，例如String.chars()，BufferedReader.lines()，Pattern.splitAsStream()，Random.ints()和BitSet.stream()。

最后，Java类库还提供了一系列API用于构建流，类库的编写者可以利用这些API来在流上实现其它聚合操作。实现Stream至少需要一个Iterator，不过如果编写者还拥有其它元数据（例如大小），类库就可以通过实现Spliterator提供一个更加高效的实现（就像JDK中所有的集合一样）。

## 比较器工厂

Comparator接口中新增了若干用于生成比较器的使用方法。

静态方法Comparator.comparing()接收一个函数（该函数返回一个实现Comparable接口的比较键值）并返回一个Comparator，它的实现十分简洁：

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

我们把这种方法称为高阶函数：以函数作为参数或是返回值的函数。我们可以使用高阶函数简化代码：

|  |
| --- |
| List<Person> people = ...  people.sort(comparing(p -> p.getLastName())); |

这段代码比“过去的代码”（一般要定义一个实现Comparator接口的匿名类）要简洁很多。但是它真正的威力在于它大大改进了可组合性（composability），举例来说，Comparator拥有一个用于逆序的默认方法，因此如果想把列表按照姓进行反序排序，我们只需要创建一个和之前一样的比较器，然后调用反序方法即可：

|  |
| --- |
| people.sort(comparing(p -> p.getLastName()).reversed()); |

与之类似，默认方法thenComparing允许你去改进一个已有的Comparator：在原比较器返回相等的结果时进行进一步比较。下面的代码演示了如何按照姓和名进行排序：

|  |
| --- |
| Comparator<Person> c =  Comparator.comparing(p -> p.getLastName())  .thenComparing(p -> p.getFirstName());  people.sort(c); |

## 可变的集合操作

集合上的流操作一般会生成一个新的值或集合，不过有时我们希望直接修改集合，所以Java类库为集合（例如Collection，List和Map）提供了一些新的方法，比如Iterable.forEach(Consumer)，Collection.removeAll(Predicate)，List.replaceAll(UnaryOperator)，List.sort(Comparator)和Map.computeIfAbsent()。除此之外，ConcurrentMap中的一些非原子方法（例如replace和putIfAbsent）被放到了Map之中。

# 小结

引入lambda表达式是Java语言的巨大进步。因为开发者每天都要使用核心类库，所以语言的演化和类库的演化是不可分割的，这样开发者能够尽可能方便的使用语言的新特性。Stream抽象作为新增类库特性的核心，提供了强大的数据聚合操作功能，并被深入整合到现有的集合类和其它的JDK类中。