

王爵nice

2018 年 03 月 30 日

Java 8 lambda 表达式10个示例

Java 8 发布于4年前，日期是2014年3月18日，这次开创性的发布在Java社区引发了不少讨论，并让大家感到激动。特性之一便是随同发布的lambda表达式，它将允许我们将行为传到函数里。在Java 8之前，如果想将行为传入函数，仅有的选择就是匿名类，需要6行代码。而定义行为最重要的那行代码，却混在中间不够突出。Lambda表达式取代了匿名类，取消了模板，允许用函数式风格编写代码。这样有时可读性更好，表达更清晰。

在Java生态系统中，函数式表达与对面向对象的全面支持是个激动人心的进步。将进一步促进并行第三方库的发展，充分利用多核CPU。

尽管业界需要时间来消化Java 8，但我认为任何严谨的Java开发者都不应忽视此次Java发布的核心特性，即lambda表达式、函数式接口、流API、默认方法和新的Date以及Time API。

作为开发人员，我发现学习和掌握lambda表达式的最佳方法就是勇于尝试，尽可能多练习lambda表达式例子。鉴于受Java 8发布的影响最大的是Java集合框架（Java Collections framework），所以最好练习流API和lambda表达式，用于对列表（Lists）和集合（Collections）数据进行提取、过滤和排序。

我一直在进行关于Java 8的写作，过去也曾分享过一些资源来帮助大家[掌握Java 8](#)。本文分享在代码中最有用的10个lambda表达式的使用方法，这些例子都短小精悍，将帮助你快速学会lambda表达式。

Java 8 lambda表达式示例

我个人对Java 8发布非常激动，尤其是lambda表达式和流API。越来越多的了解它们，我能写出更干净的代码。虽然一开始并不是这样。第一次看到用lambda表达式写出来的Java代码时，我对这种神秘的语法感到非常失望，认为它们把Java搞得不可读，但我错了。花了一天时间做了一些lambda表达式和流API示例的练习后，我开心的看到了更清晰的Java代码。这有点像学习[泛型](#)，第一次见的时候我很讨厌它。我甚至继续使用老版Java 1.4来处理集合，直到有一天，朋友跟我介绍了使用泛型的好处（才意识到它的好处）。所以基本立场就是，不要畏惧lambda表达式以及方法引用的神秘语法，做几 ▲
习，从集合类中提取、过滤数据之后，你就会喜欢上它。下面让我们开启学习Java 8 lambda表达式的

我开始使用Java 8时，首先做的就是使用lambda表达式替换匿名类，而实现Runnable接口是匿名类的最好示例。看一下Java 8之前的runnable实现方法，需要4行代码，而使用lambda表达式只需要一行代码。我们在这里做了什么呢？那就是用() -> {}代码块替代了整个匿名类。

```
// Java 8之前:
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Before Java8, too much code for too little to do");
    }
}).start();

//Java 8方式:
new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();
```

输出:

```
too much code, for too little to do
Lambda expression rocks !!
```

这个例子向我们展示了Java 8 lambda表达式的语法。你可以使用lambda写出如下代码:

```
(params) -> expression
(params) -> statement
(params) -> { statements }
```

例如，如果你的方法不对参数进行修改、重写，只是在控制台打印点东西的话，那么可以这样写:

```
() -> System.out.println("Hello Lambda Expressions");
```

如果你的方法接收两个参数，那么可以写成如下这样:

```
(int even, int odd) -> even + odd
```

如果你用过Swing API编程，你就会记得怎样写事件监听代码。这又是一个旧版本简单匿名类的经典用例，但现在可以不这样了。你可以用lambda表达式写出更好的事件监听代码，如下所示：

```
// Java 8之前:

JButton show = new JButton("Show");

show.addActionListener(new ActionListener() {

    @Override

    public void actionPerformed(ActionEvent e) {

        System.out.println("Event handling without lambda expression is boring");

    }

});

// Java 8方式:

show.addActionListener((e) -> {

    System.out.println("Light, Camera, Action !! Lambda expressions Rocks");

});
```

Java开发者经常使用匿名类的另一个地方是为 `Collections.sort()` 定制 [Comparator](#)。在Java 8中，你可以用更可读的lambda表达式换掉丑陋的匿名类。我把这个留做练习，应该不难，可以按照我在[使用lambda表达式实现 Runnable 和 ActionListener](#)的过程中的套路来做。

如果你使过几年Java，你就知道针对集合类，最常见的操作就是进行迭代，并将业务逻辑应用于各个元素，例如处理订单、交易和事件的列表。由于Java是命令式语言，Java 8之前的所有循环代码都是顺序的，即可以对其元素进行并行化处理。如果你想做并行过滤，就需要自己写代码，这并不是那么容易。通过引入lambda表达式和默认方法，将做什么和怎么做问题分开了，这意味着Java集合现在知道怎样做迭代，并可以在API层面对集合元素进行并行处理。下面的例子里，我将介绍如何在[使用lambda](#)或不使用lambda表达式的情况下迭代列表。你可以看到列表现在有了一个 `forEach()` 方法，它可以迭代所有对象，并将你的lambda代码应用在其中。

```
// Java 8之前:
List features = Arrays.asList("Lambda", "Default Method", "Stream API", "Date and Time API", "Optional", "CompletableFuture", "ForkJoinPool", "NIO.2", "JDK 8新特性");
```



```
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time A");
features.forEach(n -> System.out.println(n));
```

```
// 使用Java 8的方法引用更方便，方法引用由::双冒号操作符标示，
// 看起来像C++的作用域解析运算符
features.forEach(System.out::println);
```

输出：

```
Lambdas
Default Method
Stream API
Date and Time API
```

[列表循环](#)的最后一个例子展示了如何在Java 8中使用方法引用（method reference）。你可以看到C++里面的双冒号、范围解析操作符现在在Java 8中用来表示方法引用。

例4、使用lambda表达式和函数式接口Predicate

除了在语言层面支持函数式编程风格，Java 8也添加了一个包，叫做 `java.util.function`。它包含了很多类，用来支持Java的函数式编程。其中一个便是 `Predicate`，使用 `java.util.function.Predicate` 函数式接口以及lambda表达式，可以向API方法添加逻辑，用更少的代码支持更多的动态行为。下面是Java 8 `Predicate` 的例子，展示了过滤集合数据的多种常用方法。`Predicate`接口非常适用于做过滤。

```
public static void main(String[] args) {
    List<String> languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp");

    System.out.println("Languages which starts with J :");
    filter(languages, (str)->((String)str).startsWith("J"));

    System.out.println("Languages which ends with a ");
    filter(languages, (str)->((String)str).endsWith("a"));

    System.out.println("Print all languages :");
    filter(languages, (str)->true);
}
```



```
}

public static void filter(List<String> names, Predicate condition) {
    for(String name: names) {
        if(condition.test(name)) {
            System.out.println(name + " ");
        }
    }
}
```

输出：

```
Languages which starts with J :
Java
Languages which ends with a
Java
Scala
Print all languages :
Java
Scala
C++
Haskell
Lisp
Print no language :
Print language whose length greater than 4:
Scala
Haskell
```

// 更好的办法

```
public static void filter(List names, Predicate condition) {
    names.stream().filter((name) -> (condition.test(name))).forEach((name) -> {
        System.out.println(name + " ");
    });
}
```

可以看到，Stream API的过滤方法也接受一个**Predicate**，这意味着可以将我们定制的 filter() 方法替换成写在里面的内联代码，这就是lambda表达式的魔力。另外，**Predicate**接口也允许进行多重条件的测试，下个例子将要讲到。

例5 如何在lambda表达式中加入 Predicate

如，要得到所有以J开始，长度为四个字母的语言，可以定义两个独立的 **Predicate** 示例分别表示每一个条件，然后用 **Predicate.and()** 方法将它们合并起来，如下所示

```
// 甚至可以用and()、or()逻辑函数来合并Predicate，
// 例如要找到所有以J开始，长度为四个字母的名字，你可以合并两个Predicate并传入
Predicate<String> startsWithJ = (n) -> n.startsWith("J");
Predicate<String> fourLetterLong = (n) -> n.length() == 4;
names.stream()
    .filter(startsWithJ.and(fourLetterLong))
    .forEach((n) -> System.out.print("Name, which starts with 'J' and four letter long i
```

类似地，也可以使用 **or()** 方法。本例着重介绍了如下要点：可按需要将 **Predicate** 作为单独条件然后将其合并起来使用。简而言之，你可以以传统Java命令方式使用 **Predicate** 接口，也可以充分利用 lambda表达式达到事半功倍的效果。

例6、Java 8中使用lambda表达式的Map和Reduce示例

本例介绍最为人知的函数式编程概念map。它允许你将对象进行转换。例如在本例中，我们将 **costBeforeTax** 列表的每个元素转换成为税后的值。我们将 **x -> x*x** lambda表达式传到 **map()** 方法，后者将其应用到流中的每一个元素。然后用 **forEach()** 将列表元素打印出来。使用流API的收集器类，可以得到所有含税的开销。有 **toList()** 这样的方法将 **map** 或任何其他操作的结果合并起来。由于收集器在流上做终端操作，因此之后便不能重用流了。你甚至可以用流API的 **reduce()** 方法将所有数字合成一个，下一个例子将会讲到。

```
// 不使用lambda表达式为每个订单加上12%的税
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
for (Integer cost : costBeforeTax) {
    double price = cost + .12*cost;
    System.out.println(price);
}

// 使用lambda表达式
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
costBeforeTax.stream().map((cost) -> cost + .12*cost).forEach(System.out::println);
```

输出：

336.0
448.0
560.0
112.0
224.0
336.0
448.0
560.0

例6.2、Java 8中使用lambda表达式的Map和Reduce示例

在上个例子中，可以看到map将集合类（例如列表）元素进行转换的。还有一个 **reduce()** 函数可以将所有值合并成一个。Map和Reduce操作是函数式编程的核心操作，因为其功能，reduce 又被称为折叠操作。另外，reduce 并不是一个新的操作，你有可能已经在使用它。SQL中类似 **sum()**、**avg()** 或者 **count()** 的聚集函数，实际上就是 reduce 操作，因为它们接收多个值并返回一个值。流API定义的 **reduce()** 函数可以接受lambda表达式，并对所有值进行合并。IntStream这样的类有类似 **average()**、**count()**、**sum()** 的内建方法来做 reduce 操作，也有**mapToLong()**、**mapToDouble()** 方法来做转换。这并不会限制你，你可以用内建方法，也可以自己定义。在这个Java 8的Map Reduce 示例里，我们首先对所有价格应用 12% 的VAT，然后用 **reduce()** 方法计算总和。

```
// 为每个订单加上12%的税
// 老方法：
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double total = 0;
for (Integer cost : costBeforeTax) {
    double price = cost + .12*cost;
    total = total + price;
}
System.out.println("Total : " + total);

// 新方法：
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double bill = costBeforeTax.stream().map((cost) -> cost + .12*cost).reduce((sum, cost) ->
System.out.println("Total : " + bill);
```

输出：

例7、通过过滤创建一个String列表

过滤是Java开发者在大规模集合上的一个常用操作，而现在使用lambda表达式和流API过滤大规模数据集是惊人的简单。流提供了一个 **filter()** 方法，接受一个 **Predicate** 对象，即可以传入一个 lambda表达式作为过滤逻辑。下面的例子是用lambda表达式过滤Java集合，将帮助理解。

```
// 创建一个字符串列表，每个字符串长度大于2
List<String> filtered = strList.stream().filter(x -> x.length() > 2).collect(Collectors.toList());
System.out.printf("Original List : %s, filtered list : %s %n", strList, filtered);
```

输出：

```
Original List : [abc, , bcd, , defg, jk], filtered list : [abc, bcd, defg]
```

另外，关于 **filter()** 方法有个常见误解。在现实生活中，做过滤的时候，通常会丢弃部分，但使用 **filter()**方法则是获得一个新的列表，且其每个元素符合过滤原则。

例8、对列表的每个元素应用函数

我们通常需要对列表的每个元素使用某个函数，例如逐一乘以某个数、除以某个数或者做其它操作。这些操作都很适合用 **map()** 方法，可以将转换逻辑以lambda表达式的形式放在 **map()** 方法里，就可以对集合的各个元素进行转换了，如下所示。

```
// 将字符串换成大写并用逗号链接起来
List<String> G7 = Arrays.asList("USA", "Japan", "France", "Germany", "Italy", "U.K.", "Canada");
String G7Countries = G7.stream().map(x -> x.toUpperCase()).collect(Collectors.joining(", "));
System.out.println(G7Countries);
```

输出：

例9、复制不同的值，创建一个子列表

本例展示了如何利用流的 `distinct()` 方法来对集合进行去重。

```
// 用所有不同的数字创建一个正方形列表
List<Integer> numbers = Arrays.asList(9, 10, 3, 4, 7, 3, 4);
List<Integer> distinct = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
System.out.printf("Original List : %s, Square Without duplicates : %s %n", numbers, distinct);
```

输出：

```
Original List : [9, 10, 3, 4, 7, 3, 4], Square Without duplicates : [81, 100, 9, 16, 49]
```

例10、计算集合元素的最大值、最小值、总和以及平均值

`IntStream`、`LongStream` 和 `DoubleStream` 等流的类中，有个非常有用的方法叫做 `summaryStatistics()`。可以返回 `IntSummaryStatistics`、`LongSummaryStatistics` 或者 `DoubleSummaryStatistics`，描述流中元素的各种摘要数据。在本例中，我们用这个方法来计算列表的最大值和最小值。它也有 `getSum()` 和 `getAverage()` 方法来获得列表的所有元素的总和及平均值。

```
//获取数字的个数、最小值、最大值、总和以及平均值
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
IntSummaryStatistics stats = primes.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Highest prime number in List : " + stats.getMax());
System.out.println("Lowest prime number in List : " + stats.getMin());
System.out.println("Sum of all prime numbers : " + stats.getSum());
System.out.println("Average of all prime numbers : " + stats.getAverage());
```

输出：



Lambda表达式 vs 匿名类

既然lambda表达式即将正式取代Java代码中的匿名内部类，那么有必要对二者做一个比较分析。一个关键的不同点就是关键字 **this**。匿名类的 **this** 关键字指向匿名类，而lambda表达式的 **this** 关键字指向包围lambda表达式的类。另一个不同点是二者的编译方式。Java编译器将lambda表达式编译成类的私有方法。使用了Java 7的 **invokedynamic** 字节码指令来动态绑定这个方法。

Java 8 Lambda表达式要点

10个Java lambda表达式、流API示例

到目前为止我们看到了Java 8的10个lambda表达式，这对于新手来说是个合适的任务量，你可能需要亲自运行示例程序以便掌握。试着修改要求创建自己的例子，达到快速学习的目的。我还想建议大家使用Netbeans IDE来练习lambda表达式，它对Java 8支持良好。当把代码转换成函数式的时候，Netbeans会及时给你提示。只需跟着Netbeans的提示，就能很容易地把匿名类转换成lambda表达式。此外，如果你喜欢阅读，那么记得看一下Java 8的lambdas，实用函数式编程这本书（[Java 8 Lambdas, pragmatic functional programming](#)），作者是Richard Warburton，或者也可以看看Manning的Java 8实战（[Java 8 in Action](#)），这本书虽然还没出版，但我猜线上有第一章的免费pdf。不过，在你开始忙其它事情之前，先回顾一下Java 8的lambda表达式、默认方法和函数式接口的重点知识。

1) lambda表达式仅能放入如下代码：预定义使用了 `@Functional` 注释的函数式接口，自带一个抽象函数的方法，或者SAM（Single Abstract Method 单个抽象方法）类型。这些称为lambda表达式的目标类型，可以用作返回类型，或lambda目标代码的参数。例如，若一个方法接收Runnable、Comparable或者 Callable 接口，都有单个抽象方法，可以传入lambda表达式。类似的，如果一个方法接受声明于 `java.util.function` 包内的接口，例如 Predicate、Function、Consumer 或 Supplier，那么可以向其传lambda表达式。

2) lambda表达式内可以使用方法引用，仅当该方法不修改lambda表达式提供的参数。本例中的lambda表达式可以换为方法引用，因为这仅是一个参数相同的简单方法调用。

```
list.forEach(n -> System.out.println(n));
```

```
list.forEach((String s) -> System.out.println("*" + s + "*"));
```

事实上，可以省略这里的lambda参数的类型声明，编译器可以从列表的类属性推测出来。

3) lambda内部可以使用静态、非静态和局部变量，这称为lambda内的变量捕获。

4) Lambda表达式在Java中又称为闭包或匿名函数，所以如果有同事把它叫闭包的时候，不用惊讶。

5) Lambda方法在编译器内部被翻译成私有方法，并派发 **invokedynamic** 字节码指令来进行调用。可以使用JDK中的 **javap** 工具来反编译class文件。使用 **javap -p** 或 **javap -c -v** 命令来看一看lambda表达式生成的字节码。大致应该长这样：

```
private static java.lang.Object lambda$0(java.lang.String);
```

6) lambda表达式有个限制，那就是只能引用 **final** 或 **final** 局部变量，这就是说不能在lambda内部修改定义在域外的变量。

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});  
int factor = 2;  
primes.forEach(element -> { factor++; });
```

```
Compile time error : "local variables referenced from a lambda expression must be final o
```

另外，只是访问它而不作修改是可以的，如下所示：

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});  
int factor = 2;  
primes.forEach(element -> { System.out.println(factor*element); });
```

输出：

```
4  
6
```