学习 › Java technology

Java 8 习惯用语，第 9 部分

级联 lambda 表达式

可重用的函数有助于让代码变得非常简短，但是会不会过于简短呢?

Venkat Subramaniam
2017 年 11 月 29 日发布

0

系列内容：

＋ 此内容是该系列 11 部分中的第 9 部分： **Java 8 习惯用语**

在函数式编程中，函数既可以接收也可以返回其他函数。函数不再像传统的面向对象编程中一样，只是一个对象的*工厂*或*生成器*，它也能够创建和返回另一个函数。返回函数的函数可以变*级联 lambda 表达式*，特别值得注意的是代码非常简短。尽管此语法初看起来可能非常陌生，它有自己的用途。本文将帮助您认识级联 lambda 表达式，理解它们的性质和在代码中的用途。

神秘的语法

您是否看到过类似这样的代码段?

```
x -> y -> x > y
```

如果您很好奇“这到底是什么意思？”，那么您并不孤单。对于不熟悉使用 lambda 表达式编程的
货物正从快速行驶的卡车上一件件掉下来一样。

幸运的是，我们不会经常看到它们，但理解如何创建级联 lambda 表达式和如何在代码中理解它

高阶函数

在谈论级联 lambda 表达式之前，有必要首先理解如何创建它们。对此，我们需要回顾一下高阶
介绍）和它们在*函数分解*中的作用，函数分解是一种将复杂流程分解为更小、更简单的部分的

首先，考虑区分高阶函数与常规函数的规则：

常规函数

- 可以接收对象
- 可以创建对象
- 可以返回对象

高阶函数

- 可以接收函数
- 可以创建函数
- 可以返回函数

开发人员将匿名函数或 lambda 表达式传递给高阶函数，以让代码简短且富于表达。让我们看看

示例 1：一个接收函数的函数

在 Java™ 中，我们使用函数接口来引用 lambda 表达式和方法引用。下面这个函数接收一个对

```
public static int totalSelectedValues(List<Integer> values,
  Predicate<Integer> selector) {

  return values.stream()
    .filter(selector)
    .reduce(0, Integer::sum);
}
```

totalSelectedValues 的第一个参数是集合对象，而第二个参数是 Predicate 函数接口。 因为
(Predicate)，所以我们现在可以将一个 lambda 表达式作为第二个参数传递给 totalSelected
一个 numbers 列表中的*偶数值*求和，可以调用 totalSelectedValues，如下所示：

```
totalSelectedValues(numbers, e -> e % 2 == 0);
```

假设我们现在在 Util 类中有一个名为 isEven 的 static 方法。在此情况下，我们可以使用 is
的参数，而不传递 lambda 表达式：

```
totalSelectedValues(numbers, Util::isEven);
```

作为规则，只要一个函数接口显示为一个函数的参数的*类型*，您看到的就是一个高阶函数。

示例 2：一个返回函数的函数

函数可以接收函数、lambda 表达式或方法引用作为参数。同样地，函数也可以返回 lambda 表
回类型将是函数接口。

让我们首先看一个创建并返回 Predicate 来验证给定值是否为奇数的函数：

```
public static Predicate<Integer> createIsOdd() {
  Predicate<Integer> check = (Integer number) -> number % 2 != 0;
  return check;
}
```

为了返回一个函数，我们必须提供一个函数接口作为返回类型。在本例中，我们的函数接口是上是正确的，但它可以更加简短。 我们使用类型引用并删除临时变量来改进该代码：

```
public static Predicate<Integer> createIsOdd() {
  return number -> number % 2 != 0;
}
```

这是使用的 createIsOdd 方法的一个示例：

```
Predicate<Integer> isOdd = createIsOdd();

isOdd.test(4);
```

请注意，在 isOdd 上调用 test 会返回 false。我们也可以在 isOdd 上使用更多值来调用 test

创建可重用的函数

现在您已大体了解高阶函数和如何在代码中找到它们，我们可以考虑使用它们来让代码更加简

设想我们有两个列表 numbers1 和 numbers2。假设我们想从第一个列表中仅提取大于 50 的数，的值并*乘以 2*。

可通过以下代码实现这些目的：

```
List<Integer> result1 = numbers1.stream()
  .filter(e -> e > 50)
  .collect(toList());

List<Integer> result2 = numbers2.stream()
  .filter(e -> e > 50)
  .map(e -> e * 2)
```

```
  .collect(toList());
```

此代码很好，但您注意到它很冗长了吗？我们对检查数字是否大于 50 的 lambda 表达式使用了一个 Predicate，从而删除重复代码，让代码更富于表达：

```
Predicate<Integer> isGreaterThan50 = number -> number > 50;

List<Integer> result1 = numbers1.stream()
  .filter(isGreaterThan50)
  .collect(toList());

List<Integer> result2 = numbers2.stream()
  .filter(isGreaterThan50)
  .map(e -> e * 2)
  .collect(toList());
```

通过将 lambda 表达式存储在一个引用中，我们可以重用它，这是我们避免重复 lambda 表达式 lambda 表达式，也可以将该引用放入一个单独的方法中，而不是放在一个局部变量引用中。

现在假设我们想从列表 numbers1 中提取大于 25、50 和 75 的值。我们可以首先编写 3 个不同

```
List<Integer> valuesOver25 = numbers1.stream()
  .filter(e -> e > 25)
  .collect(toList());

List<Integer> valuesOver50 = numbers1.stream()
  .filter(e -> e > 50)
  .collect(toList());

List<Integer> valuesOver75 = numbers1.stream()
  .filter(e -> e > 75)
  .collect(toList());
```

尽管上面每个 lambda 表达式将输入与一个不同的值比较，但它们做的事情完全相同。如何以

创建和重用 lambda 表达式

尽管上一个示例中的两个 lambda 表达式相同，但上面 3 个表达式稍微不同。创建一个返回 Pr

问题。

首先，函数接口 Function<T，U> 将一个 T 类型的输入转换为 U 类型的输出。例如，下面的示例根：

```
Function<Integer, Double> sqrt = value -> Math.sqrt(value);
```

在这里，返回类型 U 可以很简单，比如 Double、String 或 Person。或者它也可以更复杂，比个函数接口。

在本例中，我们希望一个 Function 创建一个 Predicate。所以代码如下：

```
Function<Integer, Predicate<Integer>> isGreaterThan = (Integer pivot) -> {
  Predicate<Integer> isGreaterThanPivot = (Integer candidate) -> {
    return candidate > pivot;
  };

  return isGreaterThanPivot;
};
```

引用 isGreaterThan 引用了一个表示 Function<T，U>— 或更准确地讲表示 Function<Intege lambda 表达式。输入是一个 Integer，输出是一个 Predicate<Integer>。

在 lambda 表达式的主体中（外部 {} 内），我们创建了另一个引用 isGreaterThanPivot，它用。这一次，该引用是一个 Predicate 而不是 Function。最后，我们返回该引用。

isGreaterThan 是一个 lambda 表达式的引用，该表达式在调用时返回*另一个* lambda 表达式 · lambda 表达式级联关系。


```
List<Integer> valuesOver25 = numbers1.stream()
  .filter(isGreaterThan.apply(25))
  .collect(toList());

List<Integer> valuesOver50 = numbers1.stream()
  .filter(isGreaterThan.apply(50))
```

```
  .collect(toList());

List<Integer> valuesOver75 = numbers1.stream()
  .filter(isGreaterThan.apply(75))
  .collect(toList());
```

创建和重用 lambda 表达式

在 isGreaterThan 上调用 apply 会返回一个 Predicate，后者然后作为参数传递给 filter 方

保持简短的秘诀

尽管整个过程非常简单（作为示例），但是能够抽象为一个函数对于谓词更加复杂的场景来说

理解级联 lambda 表达式

保持简短的秘诀

相关主题

我们已从代码中成功删除了重复的 lambda 表达式，但 isGreaterThan 的定义看起来仍然很杂

评论

Java 8 约定来减少杂乱，让代码更简短。

我们首先重构以下代码：

```
Function<Integer, Predicate<Integer>> isGreaterThan = (Integer pivot) -> {
  Predicate<Integer> isGreaterThanPivot = (Integer candidate) -> {
    return candidate > pivot;
  };

  return isGreaterThanPivot;
};
```

可以使用类型引用来从外部和内部 lambda 表达式的参数中删除类型细节：

```
Function<Integer, Predicate<Integer>> isGreaterThan = (pivot) -> {
  Predicate<Integer> isGreaterThanPivot = (candidate) -> {
    return candidate > pivot;
  };

  return isGreaterThanPivot;
};
```

目前，我们从代码中删除了两个单词，改进不大。

接下来，我们删除多余的 `()`，以及外部 lambda 表达式中不必要的临时引用：

```
Function<Integer, Predicate<Integer>> isGreaterThan = pivot -> {
  return candidate -> {
    return candidate > pivot;
  };
};
```

代码更加简短了，但是仍然看起来有些杂乱。

可以看到内部 lambda 表达式的主体只有一行，显然 `{}` 和 `return` 是多余的。让我们删除它们

```
Function<Integer, Predicate<Integer>> isGreaterThan = pivot -> {
  return candidate -> candidate > pivot;
};
```

现在可以看到，外部 lambda 表达式的主体*也*只有一行，所以 `{}` 和 `return` 在这里也是多余的，
构：

```
Function<Integer, Predicate<Integer>> isGreaterThan =
  pivot -> candidate -> candidate > pivot;
```

现在可以看到 — 这是我们的级联 lambda 表达式。

理解级联 lambda 表达式

我们通过一个适合每个阶段的重构过程，得到了最终的代码 - 级联 lambda 表达式。在本例中，
作为参数，内部 lambda 表达式接收 `candidate` 作为参数。内部 lambda 表达式的主体同时使用
自外部范围的参数。也就是说，内部 lambda 表达式的主体同时依靠它的参数和它的*词法范围*

级联 lambda 表达式对于编写它的人非常有意义。但是对于读者呢?

看到一个只有一个向右箭头 (->) 的 lambda 表达式时，您应该知道您看到的是一个匿名函数，

一个操作或返回一个结果值。

看到一个包含两个向右箭头 (->) 的 lambda 表达式时，您看到的也是一个匿名函数，但它接受
lambda 表达式。返回的 lambda 表达式可以接受它自己的参数或者可能是空的。它可以执行一
返回另一个 lambda 表达式，但这通常有点大材小用，最好避免。

大体上讲，当您看到两个向右箭头时，可以将第一个箭头右侧的所有内容视为一个黑盒：一个
lambda 表达式。

结束语

级联 lambda 表达式不是很常见，但您应该知道如何在代码中识别和理解它们。当一个 lambda
式，而不是接受一个操作或返回一个值时，您将看到两个箭头。这种代码非常简短，但可能在
一旦您学会识别这种函数式语法，理解和掌握它就会变得容易得多。

相关主题

使用 lambda 表达式进行 Java 编程

Java 8 语言变更

Java 中的函数式编程：The Pragmatic Bookshelf，2014 年

评论

添加或订阅评论，请先登录或注册。

有新评论时提醒我

IBM **Developer**

站点反馈

我要投稿

报告滥用

第三方提示

关注微博

大学合作

选择语言

English

中文

日本語

Русский

Português (Brasil)

Español

한글

Code patterns

技术文档库

软件下载

开发者中心

订阅源

时事通讯

视频

博客

活动

社区