昵称：_Luc_
园龄：10年
荣誉：推荐博客
粉丝：2649
关注：1
+加关注

深入理解Java 8 Lambda（语言篇——lambda，方法引用，目标类型和默认方法）

作者：Lucida

- 微博：@peng_gong
- 豆瓣：@figure9

原文链接：http://zh.lucida.me/blog/java-8-lambdas-insideout-language-features

1. 深入理解Java 8 Lambda（语言篇——lambda，方法引用，目标类型和默认方法）
2. 深入理解Java 8 Lambda（类库篇——Streams API，Collector和并行）
3. 深入理解Java 8 Lambda（原理篇——Java编译器如何处理lambda）

关于

本文是深入理解Java 8 Lambda系列的第一篇，主要介绍Java 8新增的语言特性（比如lambda和方法引用），语言概念（比如目标类型和变量捕获）以及设计思路。

本文是对Brian Goetz的State of Lambda一文的翻译，那么问题来了：

为什么要写（翻译）这个系列?

1. 工作之后，我开始大量使用Java
2. 公司将会在不久的未来使用Java 8

我的评论
我的参与
最新评论
我的标签

我的标签
Lucida(1)
Luna的大学读书史(1)
面试经历(1)

随笔档案
2015年6月 (1)
2015年3月 (1)
2015年2月 (2)
2015年1月 (1)
2014年11月 (1)
2014年10月 (2)
2014年9月 (1)

3. 作为资质平庸的开发者，我需要打一点提前量，以免到时拙计
4. 为了学习Java 8（主要是其中的lambda及相关库），我先后阅读了Oracle的官方文档，Cay Horstmann（Core Java的作者）的Java 8 for the Really Impatient和Richard Warburton的Java 8 Lambdas
5. 但我感到并没有多大收获，Oracle的官方文档涉及了lambda表达式的每一个概念，但都是点到辄止；后两本书（尤其是Java 8 Lambdas）花了大量篇幅介绍Java lambda及其类库，但实质内容不多，读完了还是没有对Java lambda产生一个清晰的认识
6. 关键在于这些文章和书都没有解决我对Java lambda的困惑，比如：
   - Java 8中的lambda为什么要设计成这样？（为什么要一个lambda对应一个接口？而不是Structural Typing？）
   - lambda和匿名类型的关系是什么？lambda是匿名对象的语法糖吗？
   - Java 8是如何对lambda进行类型推导的？它的类型推导做到了什么程度？
   - Java 8为什么要引入默认方法？
   - Java编译器如何处理lambda？
   - 等等……
7. 之后我在Google搜索这些问题，然后就找到Brian Goetz的三篇关于Java lambda的文章（State of Lambda，State of Lambda libraries version和Translation of lambda），读完之后上面的问题都得到了解决
8. 为了加深理解，我决定翻译这一系列文章

警告（Caveats）

如果你不知道什么是函数式编程，或者不了解 `map` ， `filter` ， `reduce` 这些常用的高阶函数，那么你不适合阅读本文，请先学习函数式编程基础（比如这本书）。

State of Lambda by Brian Goetz

2014年5月 (1)
2014年3月 (1)
2014年2月 (2)
2013年1月 (1)
2012年2月 (1)
2011年7月 (1)
2011年4月 (1)
2011年3月 (1)
2010年12月 (1)
2010年11月 (2)
2010年8月 (2)
2010年4月 (1)
2009年9月 (2)
2008年12月 (1)
2008年8月 (1)

相册

> The high-level goal of Project Lambda is to enable programming patterns that require modeling code as data to be convenient and idiomatic in Java.

关于

本文介绍了Java SE 8中新引入的lambda语言特性以及这些特性背后的设计思想。这些特性包括：

- lambda表达式（又被成为"闭包"或"匿名方法"）
- 方法引用和构造方法引用
- 扩展的目标类型和类型推导
- 接口中的默认方法和静态方法

1. 背景

Java是一门面向对象编程语言。面向对象编程语言和函数式编程语言中的基本元素（Basic Values）都可以动态封装程序行为：面向对象编程语言使用带有方法的对象封装行为，函数式编程语言使用函数封装行为。但这个相同点并不明显，因为Java的对象往往比较"重量级"：实例化一个类型往往会涉及不同的类，并需要初始化类里的字段和方法。

不过有些Java对象只是对单个函数的封装。例如下面这个典型用例：Java API中定义了一个接口（一般被称为回调接口），用户通过提供这个接口的实例来传入指定行为，例如：

```
public interface ActionListener {
  void actionPerformed(ActionEvent e);
}
```

这里并不需要专门定义一个类来实现 `ActionListener` 接口，因为它只会在调用处被使用一次。用户一般会使用匿名类型把行为内联（inline）：

```
button.addActionListener(new ActionListener) {
  public void actionPerformed(ActionEvent e) {
    ui.dazzle(e.getModifiers());
  }
}
```

Blog(11)

CreepComics(5)

Dialer 2.0(16)

Lucida(12)

mReader(6)

Tup活动(21)

Tup活动 小图(22)

WP(56)

最新评论

1. Re:我的算法学习之路

谢谢，大佬!

--冷殇雪

2. Re:我的算法学习之路

谢谢大神文章，为我开拓眼界!

--hao_1234_1234

3. Re:我的算法学习之路

很多库都依赖于上面的模式。对于并行API更是如此，因为我们需要把待执行的代码提供给并行API，并行编程是一个非常值得研究的领域，因为在这里摩尔定律得到了重生：尽管我们没有更快的CPU核心（core），但是我们有更多的CPU核心。而串行API就只能使用有限的计算能力。

随着回调模式和函数式编程风格的日益流行，我们需要在Java中提供一种尽可能轻量级的将代码封装为数据（Model code as data）的方法。匿名内部类并不是一个好的选择，因为：

1. 语法过于冗余
2. 匿名类中的 `this` 和变量名容易使人产生误解
3. 类型载入和实例创建语义不够灵活
4. 无法捕获非 `final` 的局部变量
5. 无法对控制流进行抽象

上面的多数问题均在Java SE 8中得以解决：

- 通过提供更简洁的语法和局部作用域规则，Java SE 8彻底解决了问题1和问题2
- 通过提供更加灵活而且便于优化的表达式语义，Java SE 8绕开了问题3
- 通过允许编译器推断变量的“常量性”（finality），Java SE 8减轻了问题4带来的困扰

不过，Java SE 8的目标并非解决所有上述问题。因此捕获可变变量（问题4）和非局部控制流（问题5）并不在Java SE 8的范畴之内。（尽管我们可能会在未来提供对这些特性的支持）

2. 函数式接口（Functional interfaces）

尽管匿名内部类有着种种限制和问题，但是它有一个良好的特性，它和Java类型系统结合的十分紧密：每一个函数对象都对应一个接口类型。之所以说这个特性是良好的，是因为：

- 接口是Java类型系统的一部分
- 接口天然就拥有其运行时表示（Runtime representation）
- 接口可以通过Javadoc注释来表达一些非正式的协定（contract），例如，通过注释说明该操作应可交换（commutative）

一个小码农看的热血沸腾，然后有自灌一盆凉水。 先用一两年时间学会组装轮子（为了生存），然后在学习轮子的内部结构和制造过程（为了发展）。

--hao_1234_1234

4. Re:我的算法学习之路

@_Luc_抽象思想训练方法有很多种，在于日常生活环境以及思考为题的习惯。...

--github_white

5. Re:程序员英语学习指引

好东西,感谢博主

--freesec

阅读排行榜

1. Sublime Text 全程指引 by Lucida(243948)

2. 我的算法学习之路(157781)

3. 深入理解Java 8 Lambda（语言篇——lambda，方法引用，目标类型和默认方法）(156695)

上面提到的 `ActionListener` 接口只有一个方法，大多数回调接口都拥有这个特征：比如 `Runnable` 接口和 `Comparator` 接口。我们把这些只拥有一个方法的接口称为*函数式接口*。（之前它们被称为*SAM类型*，即*单抽象方法类型*（Single Abstract Method））

我们并不需要额外的工作来声明一个接口是函数式接口：编译器会根据接口的结构自行判断（判断过程并非简单的对接口方法计数：一个接口可能冗余的定义了一个 `Object` 已经提供的方法，比如 `toString()` ，或者定义了静态方法或默认方法，这些都不属于函数式接口方法的范畴）。不过API作者们可以通过 `@FunctionalInterface` 注解来显式指定一个接口是函数式接口（以避免无意声明了一个符合函数式标准的接口），加上这个注解之后，编译器就会验证该接口是否满足函数式接口的要求。

实现函数式类型的另一种方式是引入一个全新的*结构化*函数类型，我们也称其为"箭头"类型。例如，一个接收 `String` 和 `Object` 并返回 `int` 的函数类型可以被表示为 `(String, Object) -> int` 。我们仔细考虑了这个方式，但出于下面的原因，最终将其否定：

- 它会为Java类型系统引入额外的复杂度，并带来结构类型（Structural Type）和指名类型（Nominal Type）的混用。（Java几乎全部使用指名类型）
- 它会导致类库风格的分歧——一些类库会继续使用回调接口，而另一些类库会使用结构化函数类型
- 它的语法会变得十分笨拙，尤其在包含受检异常（checked exception）之后
- 每个函数类型很难拥有其运行时表示，这意味着开发者会受到类型擦除（erasure）的困扰和局限。比如说，我们无法对方法 `m(T->U)` 和 `m(X->Y)` 进行重载（Overload）

所以我们选择了"使用已知类型"这条路——因为现有的类库大量使用了函数式接口，通过沿用这种模式，我们使得现有类库能够直接使用lambda表达式。例如下面是Java SE 7中已经存在的函数式接口：

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- java.util.Comparator
- java.io.FileFilter
- java.beans.PropertyChangeListener

除此之外，Java SE 8中增加了一个新的包：`java.util.function`，它里面包含了常用的函数式接口，例如：

- `Predicate<T>` ——接收 `T` 对象并返回 `boolean`
- `Consumer<T>` ——接收 `T` 对象，不返回值
- `Function<T, R>` ——接收 `T` 对象，返回 `R` 对象
- `Supplier<T>` ——提供 `T` 对象（例如工厂），不接收值
- `UnaryOperator<T>` ——接收 `T` 对象，返回 `T` 对象
- `BinaryOperator<T>` ——接收两个 `T` 对象，返回 `T` 对象

除了上面的这些基本的函数式接口，我们还提供了一些针对原始类型（Primitive type）的特化（Specialization）函数式接口，例如 `IntSupplier` 和 `LongBinaryOperator`。（我们只为 `int`、`long` 和 `double` 提供了特化函数式接口，如果需要使用其它原始类型则需要进行类型转换）同样的我们也提供了一些针对多个参数的函数式接口，例如 `BiFunction<T, U, R>`，它接收 `T` 对象和 `U` 对象，返回 `R` 对象。

3. lambda表达式（lambda expressions）

匿名类型最大的问题就在于其冗余的语法。有人戏称匿名类型导致了"高度问题"（height problem）：比如前面 `ActionListener` 的例子里的五行代码中仅有一行在做实际工作。

lambda表达式是匿名方法，它提供了轻量级的语法，从而解决了匿名内部类带来的"高度问题"。

下面是一些lambda表达式：

```
(int x, int y) -> x + y
() -> 42
(String s) -> { System.out.println(s); }
```

第一个lambda表达式接收 `x` 和 `y` 这两个整形参数并返回它们的和；第二个lambda表达式不接收参数，返回整数'42'；第三个lambda表达式接收一个字符串并把它打印到控制台，不返回值。

lambda表达式的语法由参数列表、箭头符号 `->` 和函数体组成。函数体既可以是一个表达式，也可以是一个语句块：

4. Sublime Text 全程指引 by Lucida(175)

5. 90分钟实现一门编程语言——极简解释器教程(41)

- 表达式：表达式会被执行然后返回执行结果。
- 语句块：语句块中的语句会被依次执行，就像方法中的语句一样——
  - `return` 语句会把控制权交给匿名方法的调用者
  - `break` 和 `continue` 只能在循环中使用
  - 如果函数体有返回值，那么函数体内部的每一条路径都必须返回值

表达式函数体适合小型lambda表达式，它消除了 `return` 关键字，使得语法更加简洁。

lambda表达式也会经常出现在嵌套环境中，比如说作为方法的参数。为了使lambda表达式在这些场景下尽可能简洁，我们去除了不必要的分隔符。不过在某些情况下我们也可以把它分为多行，然后用括号包起来，就像其它普通表达式一样。

下面是一些出现在语句中的lambda表达式：

```
FileFilter java = (File f) -> f.getName().endsWith("*.java");

String user = doPrivileged(() -> System.getProperty("user.name"));

new Thread(() -> {
  connectToService();
  sendNotification();
}).start();
```

4. 目标类型（Target typing）

需要注意的是，函数式接口的名称并不是lambda表达式的一部分。那么问题来了，对于给定的lambda表达式，它的类型是什么？答案是：它的类型是由其上下文推导而来。例如，下面代码中的lambda表达式类型是 `ActionListener` ：

```
ActionListener l = (ActionEvent e) -> ui.dazzle(e.getModifiers());
```

这就意味着同样的lambda表达式在不同上下文里可以拥有不同的类型：

```
Callable<String> c = () -> "done";
```

```
PrivilegedAction<String> a = () -> "done";
```

第一个lambda表达式 `() -> "done"` 是 `Callable` 的实例，而第二个lambda表达式则是 `PrivilegedAction` 的实例。

编译器负责推导lambda表达式的类型。它利用lambda表达式所在上下文**所期待的类型**进行推导，这个**被期待的类型**被称为*目标类型*。lambda表达式只能出现在目标类型为函数式接口的上下文中。

当然，lambda表达式对目标类型也是有要求的。编译器会检查lambda表达式的类型和目标类型的方法签名（method signature）是否一致。当且仅当下面所有条件均满足时，lambda表达式才可以被赋给目标类型 `T`：

- `T` 是一个函数式接口
- lambda表达式的参数和 `T` 的方法参数在数量和类型上一一对应
- lambda表达式的返回值和 `T` 的方法返回值相兼容（Compatible）
- lambda表达式内所抛出的异常和 `T` 的方法 `throws` 类型相兼容

由于目标类型（函数式接口）已经"知道"lambda表达式的形式参数（Formal parameter）类型，所以我们没有必要把已知类型再重复一遍。也就是说，lambda表达式的参数类型可以从目标类型中得出：

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

在上面的例子里，编译器可以推导出 `s1` 和 `s2` 的类型是 `String`。此外，当lambda的参数只有一个而且它的类型可以被推导得知时，该参数列表外面的括号可以被省略：

```
FileFilter java = f -> f.getName().endsWith(".java");

button.addActionListener(e -> ui.dazzle(e.getModifiers()));
```

这些改进进一步展示了我们的设计目标："不要把高度问题转化成宽度问题。"我们希望语法元素能够尽可能的少，以便代码的读者能够直达lambda表达式的核心部分。

lambda表达式并不是第一个拥有上下文相关类型的Java表达式：泛型方法调用和"菱形"构造器调用也通过目标类型来进行类型推导：

```
List<String> ls = Collections.emptyList();
List<Integer> li = Collections.emptyList();

Map<String, Integer> m1 = new HashMap<>();
Map<Integer, String> m2 = new HashMap<>();
```

5. 目标类型的上下文（Contexts for target typing）

之前我们提到lambda表达式智能出现在拥有目标类型的上下文中。下面给出了这些带有目标类型的上下文：

- 变量声明
- 赋值
- 返回语句
- 数组初始化器
- 方法和构造方法的参数
- lambda表达式函数体
- 条件表达式（`? :`）
- 转型（Cast）表达式

在前三个上下文（变量声明、赋值和返回语句）里，目标类型即是被赋值或被返回的类型：

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

public Runnable toDoLater() {
  return () -> {
    System.out.println("later");
  }
}
```

数组初始化器和赋值类似，只是这里的“变量”变成了数组元素，而类型是从数组类型中推导得知：

```
filterFiles(new FileFilter[] {
              f -> f.exists(), f -> f.canRead(), f -> f.getName().startsWith("q")
            });
```

方法参数的类型推导要相对复杂些：目标类型的确认会涉及到其它两个语言特性：重载解析（Overload resolution）和参数类型推导（Type argument inference）。

重载解析会为一个给定的方法调用（method invocation）寻找最合适的方法声明（method declaration）。由于不同的声明具有不同的签名，当lambda表达式作为方法参数时，重载解析就会影响到lambda表达式的目标类型。编译器会通过它所得之的信息来做出决定。如果lambda表达式具有**显式类型**（参数类型被显式指定），编译器就可以直接 使用lambda表达式的返回类型；如果lambda表达式具有**隐式类型**（参数类型被推导而知），重载解析则会忽略lambda表达式函数体而只依赖lambda表达式参数的数量。

如果在解析方法声明时存在二义性（ambiguous），我们就需要利用转型（cast）或显式lambda表达式来提供更多的类型信息。如果lambda表达式的返回类型依赖于其参数的类型，那么lambda表达式函数体有可能可以给编译器提供额外的信息，以便其推导参数类型。

```
List<Person> ps = ...
Stream<String> names = ps.stream().map(p -> p.getName());
```

在上面的代码中，`ps` 的类型是 `List<Person>`，所以 `ps.stream()` 的返回类型是 `Stream<Person>`。`map()` 方法接收一个类型为 `Function<T, R>` 的函数式接口，这里 `T` 的类型即是 `Stream` 元素的类型，也就是 `Person`，而 `R` 的类型未知。由于在重载解析之后lambda表达式的目标类型仍然未知，我们就需要推导 `R` 的类型：通过对lambda表达式函数体进行类型检查，我们发现函数体返回 `String`，因此 `R` 的类型是 `String`，因而 `map()` 返回 `Stream<String>`。绝大多数情况下编译器都能解析出正确的类型，但如果碰到无法解析的情况，我们则需要：

- 使用显式lambda表达式（为参数 `p` 提供显式类型）以提供额外的类型信息
- 把lambda表达式转型为 `Function<Person, String>`
- 为泛型参数 `R` 提供一个实际类型。（`.<String>map(p -> p.getName())`）

lambda表达式本身也可以为它自己的函数体提供目标类型，也就是说lambda表达式可以通过外部目标类型推导出其内部的返回类型，这意味着我们可以方便的编写一个返回函数的函数：

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

类似的，条件表达式可以把目标类型“分发”给其子表达式：

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

最后，转型表达式（Cast expression）可以显式提供lambda表达式的类型，这个特性在无法确认目标类型时非常有用：

```
// Object o = () -> { System.out.println("hi"); }; 这段代码是非法的
Object o = (Runnable) () -> { System.out.println("hi"); };
```

除此之外，当重载的方法都拥有函数式接口时，转型可以帮助解决重载解析时出现的二义性。

目标类型这个概念不仅仅适用于lambda表达式，泛型方法调用和“菱形”构造方法调用也可以从目标类型中受益，下面的代码在Java SE 7是非法的，但在Java SE 8中是合法的：

```
List<String> ls = Collections.checkedList(new ArrayList<>(), String.class);

Set<Integer> si = flag ? Collections.singleton(23) : Collections.emptySet();
```

6. 词法作用域（Lexical scoping）

在内部类中使用变量名（以及 `this` ）非常容易出错。内部类中通过继承得到的成员（包括来自 `Object` 的方法）可能会把外部类的成员掩盖（shadow），此外未限定（unqualified）的 `this` 引用会指向内部类自己而非外部类。

相对于内部类，lambda表达式的语义就十分简单：它不会从超类（supertype）中继承任何变量名，也不会引入一个新的作用域。lambda表达式基于词法作用域，也就是说lambda表达式函数体里面的变量和它外部环境的变量具有相同的语义（也包括lambda表达式的形式参数）。此外，'this'关键字及其引用在lambda表达式内部和外部也拥有相同的语义。

为了进一步说明词法作用域的优点，请参考下面的代码，它会把 `"Hello, world!"` 打印两遍：

```
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();
  }
}
```

与之相类似的内部类实现则会打印出类似 `Hello$1@5b89a773` 和 `Hello$2@537a7706` 之类的字符串，这往往会使开发者大吃一惊。

基于词法作用域的理念，lambda表达式不可以掩盖任何其所在上下文中的局部变量，它的行为和那些拥有参数的控制流结构（例如 `for` 循环和 `catch` 从句）一致。

个人补充：这个说法很拗口，所以我在这里加一个例子以演示词法作用域：

```
int i = 0;
int sum = 0;
for (int i = 1; i < 10; i += 1) { //这里会出现编译错误，因为i已经在for循环外部声明过了
  sum += i;
}
```

7. 变量捕获（Variable capture）

在Java SE 7中，编译器对内部类中引用的外部变量（即捕获的变量）要求非常严格：如果捕获的变量没有被声明为 `final` 就会产生一个编译错误。我们现在放宽了这个限制——对于lambda表达式和内部类，我们允许在其中捕获那些符合*有效只读*（Effectively final）的局部变量。

简单的说，如果一个局部变量在初始化后从未被修改过，那么它就符合有效只读的要求，换句话说，加上 `final` 后也不会导致编译错误的局部变量就是有效只读变量。

```
Callable<String> helloCallable(String name) {
  String hello = "Hello";
  return () -> (hello + ", " + name);
}
```

对 `this` 的引用，以及通过 `this` 对未限定字段的引用和未限定方法的调用在本质上都属于使用 `final` 局部变量。包含此类引用的lambda表达式相当于捕获了 `this` 实例。在其它情况下，lambda对象不会保留任何对 `this` 的引用。

这个特性对内存管理是一件好事：内部类实例会一直保留一个对其外部类实例的强引用，而那些没有捕获外部类成员的lambda表达式则不会保留对外部类实例的引用。要知道内部类的这个特性往往会造成内存泄露。

尽管我们放宽了对捕获变量的语法限制，但试图修改捕获变量的行为仍然会被禁止，比如下面这个例子就是非法的：

```
int sum = 0;
list.forEach(e -> { sum += e.size(); });
```

为什么要禁止这种行为呢？因为这样的lambda表达式很容易引起race condition。除非我们能够强制（最好是在编译时）这样的函数不能离开其当前线程，但如果这么做了可能会导致更多的问题。简而言之，lambda表达式对*值*封闭，对*变量*开放。

个人补充：lambda表达式对*值*封闭，对*变量*开放的原文是：lambda expressions close over *values*, not *variables*，我在这里增加一个例子以说明这个特性：

```
int sum = 0;
list.forEach(e -> { sum += e.size(); }); // Illegal, close over values

List<Integer> aList = new List<>();
list.forEach(e -> { aList.add(e); }); // Legal, open over variables
```

lambda表达式不支持修改捕获变量的另一个原因是我们可以使用更好的方式来实现同样的效果：使用规约（reduction）。 `java.util.stream` 包提供了各种通用的和专用的规约操作（例如 `sum` 、 `min` 和 `max` ），就上面的例子而言，我们可以使用规约操作（在串行和并行下都是安全的）来代替 `forEach` ：

```
int sum = list.stream()
              .mapToInt(e -> e.size())
              .sum();
```

`sum()` 等价于下面的规约操作：

```
int sum = list.stream()
              .mapToInt(e -> e.size())
```

```
                .reduce(0 , (x, y) -> x + y);
```

规约需要一个初始值（以防输入为空）和一个操作符（在这里是加号），然后用下面的表达式计算结果：

```
0 + list[0] + list[1] + list[2] + ...
```

规约也可以完成其它操作，比如求最小值、最大值和乘积等等。如果操作符具有可结合性（associative），那么规约操作就可以容易的被并行化。所以，与其支持一个本质上是并行而且容易导致race condition的操作，我们选择在库中提供一个更加并行友好且不容易出错的方式来进行累积（accumulation）。

8. 方法引用（Method references）

lambda表达式允许我们定义一个匿名方法，并允许我们以函数式接口的方式使用它。我们也希望能够在*已有的*方法上实现同样的特性。

方法引用和lambda表达式拥有相同的特性（例如，它们都需要一个目标类型，并需要被转化为函数式接口的实例），不过我们并不需要为方法引用提供方法体，我们可以直接通过方法名称引用已有方法。

以下面的代码为例，假设我们要按照 `name` 或 `age` 为 `Person` 数组进行排序：

```
class Person {
  private final String name;
  private final int age;

  public int getAge() { return age; }
  public String getName() {return name; }
  ...
}

Person[] people = ...
Comparator<Person> byName = Comparator.comparing(p -> p.getName());
Arrays.sort(people, byName);
```

在这里我们可以用方法引用代替lambda表达式：

```
Comparator<Person> byName = Comparator.comparing(Person::getName);
```

这里的 `Person::getName` 可以被看作为lambda表达式的简写形式。尽管方法引用不一定（比如在这个例子里）会把语法变的更紧凑，但它拥有更明确的语义——如果我们想要调用的方法拥有一个名字，我们就可以通过它的名字直接调用它。

因为函数式接口的方法参数对应于隐式方法调用时的参数，所以被引用方法签名可以通过放宽类型，装箱以及组织到参数数组中的方式对其参数进行操作，就像在调用实际方法一样：

```
Consumer<Integer> b1 = System::exit;     // void exit(int status)
Consumer<String[]> b2 = Arrays:sort;     // void sort(Object[] a)
Consumer<String> b3 = MyProgram::main;   // void main(String... args)
Runnable r = Myprogram::mapToInt         // void main(String... args)
```

9. 方法引用的种类（Kinds of method references）

方法引用有很多种，它们的语法如下：

- 静态方法引用：`ClassName::methodName`
- 实例上的实例方法引用：`instanceReference::methodName`
- 超类上的实例方法引用：`super::methodName`
- 类型上的实例方法引用：`ClassName::methodName`
- 构造方法引用：`Class::new`
- 数组构造方法引用：`TypeName[]::new`

对于静态方法引用，我们需要在类名和方法名之间加入 `::` 分隔符，例如 `Integer::sum` 。

对于具体对象上的实例方法引用，我们则需要在对象名和方法名之间加入分隔符：

```
Set<String> knownNames = ...
Predicate<String> isKnown = knownNames::contains;
```

这里的隐式lambda表达式（也就是实例方法引用）会从 `knownNames` 中捕获 `String` 对象，而它的方法体则会通过 `Set.contains` 使用该 `String` 对象。

有了实例方法引用，在不同函数式接口之间进行类型转换就变的很方便：

```
Callable<Path> c = ...
Privileged<Path> a = c::call;
```

引用任意对象的实例方法则需要在实例方法名称和其所属类型名称间加上分隔符：

```
Function<String, String> upperfier = String::toUpperCase;
```

这里的隐式lambda表达式（即 `String::toUpperCase` 实例方法引用）有一个 `String` 参数，这个参数会被 `toUpperCase` 方法使用。

如果类型的实例方法是泛型的，那么我们就需要在 `::` 分隔符前提供类型参数，或者（多数情况下）利用目标类型推导出其类型。

需要注意的是，静态方法引用和类型上的实例方法引用拥有一样的语法。编译器会根据实际情况做出决定。

一般我们不需要指定方法引用中的参数类型，因为编译器往往可以推导出结果，但如果需要我们也可以显式在 `::` 分隔符之前提供参数类型信息。

和静态方法引用类似，构造方法也可以通过 `new` 关键字被直接引用：

```
SocketImplFactory factory = MySocketImpl::new;
```

如果类型拥有多个构造方法，那么我们就会通过目标类型的方法参数来选择最佳匹配，这里的选择过程和调用构造方法时的选择过程是一样的。

如果待实例化的类型是泛型的，那么我们可以在类型名称之后提供类型参数，否则编译器则会依照"菱形"构造方法调用时的方式进行推导。

数组的构造方法引用的语法则比较特殊，为了便于理解，你可以假想存在一个接收 `int` 参数的数组构造方法。参考下面的代码：

```
IntFunction<int[]> arrayMaker = int[]::new;
int[] array = arrayMaker.apply(10) // 创建数组 int[10]
```

10. 默认方法和静态接口方法（Default and static interface methods）

lambda表达式和方法引用大大提升了Java的表达能力（expressiveness），不过为了使把*代码即数据*（code-as-data）变的更加容易，我们需要把这些特性融入到已有的库之中，以便开发者使用。

Java SE 7时代为一个已有的类库增加功能是非常困难的。具体的说，接口在发布之后就已经被定型，除非我们能够一次性更新所有该接口的实现，否则向接口添加方法就会破坏现有的接口实现。*默认方法*（之前被称为*虚拟扩展方法*或*守护方法*）的目标即是解决这个问题，使得接口在发布之后仍能被逐步演化。

这里给出一个例子，我们需要在标准集合API中增加针对lambda的方法。例如 `removeAll` 方法应该被泛化为接收一个函数式接口 `Predicate` ，但这个新的方法应该被放在哪里呢？我们无法直接在 `Collection` 接口上新增方法——不然就会破坏现有的 `Collection` 实现。我们倒是可以在 `Collections` 工具类中增加对应的静态方法，但这样就会把这个方法置于"二等公民"的境地。

*默认方法*利用面向对象的方式向接口增加新的行为。它是一种新的方法：接口方法可以是*抽象的*或是*默认的*。默认方法拥有其默认实现，实现接口的类型通过继承得到该默认实现（如果类型没有覆盖该默认实现）。此外，默认方法不是抽象方法，所以我们可以放心的向函数式接口里增加默认方法，而不用担心函数式接口的单抽象方法限制。

下面的例子展示了如何向 `Iterator` 接口增加默认方法 `skip` ：

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();

  default void skip(int i) {
    for ( ; i > 0 && hasNext(); i -= 1) next();
  }
}
```

根据上面的 `Iterator` 定义，所有实现 `Iterator` 的类型都会自动继承 `skip` 方法。在使用者的眼里， `skip` 不过是接口新增的一个虚拟方法。在没有覆盖 `skip` 方法的 `Iterator` 子类实例上调用 `skip` 会执行 `skip` 的默认实现：调用 `hasNext` 和 `next` 若干次。子类可以通过覆盖 `skip` 来提供更好的实现——比如直接移动游标（cursor），或是提供为操作提供原子性（Atomicity）等。

当接口继承其它接口时，我们既可以为它所继承而来的抽象方法提供一个默认实现，也可以为它继承而来的默认方法提供一个新的实现，还可以把它继承而来的默认方法重新抽象化。

除了默认方法，Java SE 8还在允许在接口中定义*静态*方法。这使得我们可以从接口直接调用和它相关的辅助方法（Helper method），而不是从其它的类中调用（之前这样的类往往以对应接口的复数命名，例如 `Collections` ）。比如，我们一般需要使用静态辅助方法生成实现 `Comparator` 的比较器，在Java SE 8中我们可以直接把该静态方法定义在 `Comparator` 接口中：

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

11. 继承默认方法（Inheritance of default methods）

和其它方法一样，默认方法也可以被继承，大多数情况下这种继承行为和我们所期待的一致。不过，当类型或者接口的超类拥有多个具有相同签名的方法时，我们就需要一套规则来解决这个冲突：

- 类的方法（class method）声明优先于接口默认方法。无论该方法是具体的还是抽象的。
- 被其它类型所覆盖的方法会被忽略。这条规则适用于超类型共享一个公共祖先的情况。

为了演示第二条规则，我们假设 `Collection` 和 `List` 接口均提供了 `removeAll` 的默认实现，然后 `Queue` 继承并覆盖了 `Collection` 中的默认方法。在下面的 `implement` 从句中， `List` 中的方法声明会优先于 `Queue` 中的方法声明：

```
class LinkedList<E> implements List<E>, Queue<E> { ... }
```

当两个独立的默认方法相冲突或是默认方法和抽象方法相冲突时会产生编译错误。这时程序员需要显式覆盖超类方法。一般来说我们会定义一个默认方法，然后在其中显式选择超类方法：

```
interface Robot implements Artist, Gun {
  default void draw() { Artist.super.draw(); }
}
```

`super` 前面的类型必须是有定义或继承默认方法的类型。这种方法调用并不只限于消除命名冲突——我们也可以在其它场景中使用它。

最后，接口在 `inherits` 和 `extends` 从句中的声明顺序和它们被实现的顺序无关。

12. 融会贯通（Putting it together）

我们在设计lambda时的一个重要目标就是新增的语言特性和库特性能够无缝结合（designed to work together）。接下来，我们通过一个实际例子（按照姓对名字列表进行排序）来演示这一点：

比如说下面的代码：

```
List<Person> people = ...
Collections.sort(people, new Comparator<Person>() {
  public int compare(Person x, Person y) {
    return x.getLastName().compareTo(y.getLastName());
  }
})
```

冗余代码实在太多了！

有了lambda表达式，我们可以去掉冗余的匿名类：

```
Collections.sort(people,
                 (Person x, Person y) -> x.getLastName().compareTo(y.getLastName()));
```

尽管代码简洁了很多，但它的抽象程度依然很差：开发者仍然需要进行实际的比较操作（而且如果比较的值是原始类型那么情况会更糟），所以我们要借助 `Comparator` 里的 `comparing` 方法实现比较操作：

```
Collections.sort(people, Comparator.comparing((Person p) -> p.getLastName()));
```

在类型推导和静态导入的帮助下，我们可以进一步简化上面的代码：

```
Collections.sort(people, comparing(p -> p.getLastName()));
```

我们注意到这里的lambda表达式实际上是 `getLastName` 的代理（forwarder），于是我们可以用方法引用代替它：

```
Collections.sort(people, comparing(Person::getLastName));
```

最后，使用 `Collections.sort` 这样的辅助方法并不是一个好主意：它不但使代码变的冗余，也无法为实现 `List` 接口的数据结构提供特定（specialized）的高效实现，而且由于 `Collections.sort` 方法不属于 `List` 接口，用户在阅读 `List` 接口的文档时不会察觉在另外的 `Collections` 类中还有一个针对 `List` 接口的排序（ `sort()` ）方法。

默认方法可以有效的解决这个问题，我们为 `List` 增加默认方法 `sort()` ，然后就可以这样调用：

```
people.sort(comparing(Person::getLastName));;
```

此外，如果我们为 `Comparator` 接口增加一个默认方法 `reversed()` （产生一个逆序比较器），我们就可以非常容易的在前面代码的基础上实现降序排序。

```
people.sort(comparing(Person::getLastName).reversed());;
```

13. 小结（Summary）

Java SE 8提供的新语言特性并不算多——lambda表达式，方法引用，默认方法和静态接口方法，以及范围更广的类型推导。但是把它们结合在一起之后，开发者可以编写出更加清晰简洁的代码，类库编写者可以编写更加强大易用的并行类库。

未完待续——

下篇：深入理解Java 8 Lambda（类库篇——Streams API，Collector和并行）

作者：Lucida

- 微博：@peng_gong
- 豆瓣：@figure9

原文链接：http://zh.lucida.me/blog/java-8-lambdas-insideout-language-features

Luc
关注 - 1
粉丝 - 2649
荣誉：推荐博客
+加关注

« 上一篇：Top Coder算法题目浏览器
» 下一篇：精益技术简历之道——改善技术简历的47条原则

posted @ 2014-10-24 15:22 _Luc_ 阅读(156704) 评论(15) 编辑 收藏

评论列表

#1楼 2014-10-24 15:51 CaiYongji

大哥真棒！就是想弱弱的问下，T对象和R对象有什么区别。我之前写反射的时候想把传入的类写的灵活点，封装了好久也没封好。逻辑有，但是没用JAVA表达出来。这个T和R到底有什么区别啊。

支持(4) 反对(0)

#2楼 2014-10-26 14:11 zetee

JAVA8 比.net 3.5晚了8年，从语言成面上讲java 落后太大了。互相借鉴也是必要的。

支持(5) 反对(0)

#3楼 2014-10-26 17:00 liujf

依稀记得.net早就有这些特新了

支持(2) 反对(0)

#4楼 2014-10-26 18:24 Netsharp

.net中lambda出来早，我尽量不用，代码不好维护

java中ate、Decimal这些基本的数据类型的处理让人无法忍受

支持(0) 反对(0)

#5楼 2014-10-26 18:25 Netsharp

java中Date、Decimal这些基本的数据类型的处理让人无法忍受，把基本问题解决好，比提供lambda强多了

支持(0) 反对(1)

#6楼 2014-10-26 20:59 jipinheiniu

@ takeurhand
JAVA中的lambda本身是匿名内部内的一个延伸 区别在于 lambda的类型只能是函数式接口 上面所说的T和R实际上是泛型 至于到底是何种类型拒绝于具体的逻辑 NET中虽然早就有lambda这个东西 但是和java的实现方式完全不一样 NET中的lambda是"委托"的一个延伸 NET的事件和多线程也是基于委托 而JAVA的lambda是匿名内部类的延伸

支持(0) 反对(0)

#7楼 2014-10-26 21:05 jipinheiniu

@ Netsharp
java中抛弃了结构（struct）类型 这些基本类型和NET中区别并不大都是值类型 区别在于NET使用struct把基本数据类型封装了一下 所以就使得NET中基本类型虽然是值类型 但是具有了面向对象的一些特征 所以使用起来很方便 但是java为8个基本类型添加了对应的封装类 只是使用习惯的问题

支持(0) 反对(0)

#8楼 2014-10-28 10:45 百世经伦一页书

.Net 2.0的东东

支持(0) 反对(0)

#9楼 2016-06-20 15:10 xingqx

@ jipinheiniu
.Net 的Lambda是.Net3.5中的特性，早在2008年都已经用上了。微软MSDN上解释是：
Lambda 表达式是一种可用于创建委托或表达式目录树类型的匿名函数。 通过使用 lambda 表达式，可以写入可作为参数传递或作为函数调用值返回的本地函数。 Lambda 表达式对于编写 LINQ 查询表达式特别有用。
若要创建 Lambda 表达式，需要在 Lambda 运算符 => 左侧指定输入参数（如果有），然后在另一侧输入表达式或语句块。 例如，lambda 表达式 x => x * x 指定名为 x 的参数并返回 x 的平方值。

支持(0) 反对(0)

#10楼 2016-08-05 11:35 suyisong

期待后几篇的翻译

支持(0) 反对(0)

#11楼 2016-09-22 13:45 Ggx的一天

java的lamdba和.net的不一样

支持(0) 反对(0)

#12楼 2017-02-21 16:43 遁地龙卷风

@ CaiYongji
T 对象是第一个参数
R 对象是返回结果

这表示参数和返回值是需要通过上下文进行推导的

支持(0) 反对(0)

#13楼 2017-03-05 20:48 被罚站的树

mark

支持(0) 反对(0)

#14楼 2017-07-31 13:53 烟溪秋叶

.net中的Lambda远比这个要强大，性能也没的说；java在语言层面落后太远啦！希望能加快发展速度！

支持(0) 反对(0)

#15楼 2018-07-20 10:38 RussellJX

@ CaiYongji
T and R is just a symbol for coder to read.
you can define a variable like:
String username="TOM";//line 1
or
String a = "TOM"//line 2
Line 1 and 2 are same

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· 为提高交付率不惜一切 特斯拉减少全系车身配色选项