链滴

Java 8 新特性

作者：shark

原文链接：https://ld246.com/article/1590551618305

来源网站：链滴

Java 8 新特性

1.lambda 表达式

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。

Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。

使用 Lambda 表达式可以使代码变的更加简洁紧凑。

基本的语法格式为：

```
(parameters) -> expression
或
(parameters) ->{ statements; }
```

- 可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。
- 可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。
- 可选的大括号：如果主体包含了一个语句，就不需要使用大括号。
- 可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定明表式返回了一个数值。

简单实例：

```
// 1. 不需要参数,返回值为 5
() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值
x -> 2 * x
```

```
// 3. 接受2个参数(数字),并返回他们的差值
(x, y) -> x – y

// 4. 接收2个int型整数,返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)
(String s) -> System.out.print(s)
```

给出下面这个例子：

```
public class LambdaTest1 {
    public static void main(String[] args) {
    //lambda表达式中，写的是MathOperation接口对应的实现
        MathOperation operation = (a,b)->{
            return a+b;
        };
        //调用具体的实现方法
        System.out.println(operation.operation(3, 4));

        MathOperation operation2 = (int a,int b)->{
            return a*b;
        }
        System.out.println(operation2.operation(3,4));

        MathOperation operation3 = (int a,int b)->{
            return a-b;
        };
        System.out.println(operation3.operation(3,4));

        GreetingService greetingService = message -> System.out.println(message);

        greetingService.sayMessage("Hello World!");
    }
}

interface MathOperation {
    int operation(int a, int b);
}
interface GreetingService {
    void sayMessage(String message);
}
```

执行的结果如图：

```
"E:\Program Files\Java\jdk1.8.0_212\bin\java.exe" ...
7
12
-1
Hello World!

Process finished with exit code 0
```

显然，使用Lambda表达式能够使得代码更加简洁。

使用 Lambda 表达式需要注意以下两点：

1. Lambda 表达式主要用来定义行内执行的方法类型接口，例如，一个简单方法接口。在上面例子中我们使用各种类型的Lambda表达式来定义MathOperation接口的方法。然后我们定义了sayMessag的执行。

2. Lambda 表达式免去了使用匿名方法的麻烦，并且给予Java简单但是强大的函数化的编程能力。

变量作用域：

Lambda 表达式只能引用标记了 final 的外层局部变量，这就是说不能在 lambda 内部修改定义在域的局部变量，否则会编译错误。

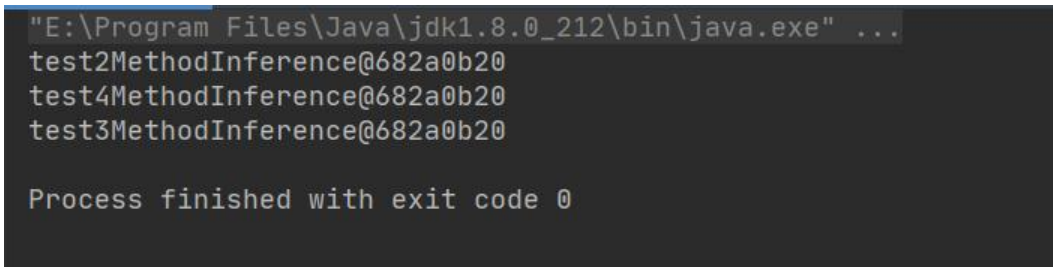
2.方法引用

- 方法引用通过方法的名字来指向一个方法。
- 方法引用可以使语言的构造更紧凑简洁，减少冗余代码。
- 方法引用使用一对冒号 :: 。

```
import java.util.Arrays;
import java.util.List;

public class MethodInference {

    public static MethodInference test1(final MyFunctionalInterface<MethodInference> myFun
tionalInterface){
        return myFunctionalInterface.get();
    }

    public static void test2(final MethodInference methodInference){
        System.out.println("test2"+methodInference);
    }

    public void test3(final MethodInference methodInference){
        System.out.println("test3"+methodInference);
    }

    public void test4(){
        System.out.println("test4"+this.toString());
    }

    public static void main(String[] args) {
//        构造器引用
        final MethodInference methodInference = MethodInference.test1(MethodInference::new)

        final List<MethodInference> methodInferences = Arrays.asList(methodInference);

//        静态方法引用
        methodInferences.forEach(MethodInference::test2);
//        特定类的任意对象的方法引用
        methodInferences.forEach(MethodInference::test4);
//        特定对象的方法引用
```

```
        final MethodInference methodInference1 = MethodInference.test1(MethodInference::ne
);
        methodInferences.forEach(methodInference1::test3);
    }


}

//使用此注解声明一个函数式接口
@FunctionalInterface
interface MyFunctionalInterface<T>{
    T get();
}
```

运行结果如下，是不是很神奇？



3.函数式接口

函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接
。
函数式接口可以被隐式转换为 lambda 表达式。
Lambda 表达式和方法引用（实际上也可认为是Lambda表达式）上。
如定义了一个函数式接口如下：

```
@FunctionalInterface
interface GreetingService
{
    void sayMessage(String message);
}
```

那么就可以使用Lambda表达式来表示该接口的一个实现(注：JAVA 8 之前一般是用匿名类实现的)：

```
GreetingService greetService1 = message -> System.out.println("Hello " + message);
```

函数式接口可以对现有的函数友好地支持 lambda。
JDK 1.8 新增加的函数接口：java.util.function
java.util.function 它包含了很多类，用来支持 Java的 函数式编程

4.默认方法

Java 8 新增了接口的默认方法。

简单说，默认方法就是接口可以有实现方法，而且不需要实现类去实现其方法。

我们只需在方法名前面加个 default 关键字即可实现默认方法。

语法

默认方法语法格式如下：

```
public interface Vehicle {
   default void print(){
      System.out.println("我是一辆车!");
   }
}
```

静态默认方法

```
public interface Vehicle {
   default void print(){
      System.out.println("我是一辆车!");
   }
    // 静态方法
   static void blowHorn(){
      System.out.println("按喇叭!!!");
   }
}
```

5.Java 8 Stream

ava 8 API添加了一个新的抽象称为流Stream，可以让你以一种声明的方式处理数据。

Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达高阶抽象。

Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

这种风格将要处理的元素集合看作一种流， 流在管道中传输， 并且可以在管道的节点上进行处理，如筛选， 排序，聚合等。

元素流在管道中经过中间操作（intermediate operation）的处理，最后由最终操作(terminal operaton)得到前面处理的结果。

什么是 Stream?

Stream（流）是一个来自数据源的元素队列并支持聚合操作

- 元素是特定类型的对象，形成一个队列。 Java中的Stream并不会存储元素，而是按需计算。
- 数据源 流的来源。 可以是集合，数组，I/O channel， 产生器generator 等。
- 聚合操作 类似SQL语句一样的操作， 比如filter, map, reduce, find, match, sorted等。

和以前的Collection操作不同， Stream操作还有两个基础的特征：

● Pipelining: 中间操作都会返回流对象本身。 这样多个操作可以串联成一个管道， 如同流式风格（fl
ent style）。 这样做可以对操作进行优化， 比如延迟执行(laziness)和短路(short-circuiting)。

● 内部迭代： 以前对集合遍历都是通过Iterator或者For-Each的方式, 显式的在集合外部进行迭代，
叫做外部迭代。 Stream提供了内部迭代的方式， 通过访问者模式(Visitor)实现。

生成流

在 Java 8 中, 集合接口有两个方法来生成流：

● stream() − 为集合创建串行流。

● parallelStream() − 为集合创建并行流。

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList
));
```

forEach

Stream 提供了新的方法 'forEach' 来迭代流中的每个数据。以下代码片段使用 forEach 输出了10
随机数：

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

map

map 方法用于映射每个元素到对应的结果，以下代码片段使用 map 输出了元素对应的平方数：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
// 获取对应的平方数
List<Integer> squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList())
```

map

map 方法用于映射每个元素到对应的结果，以下代码片段使用 map 输出了元素对应的平方数：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
// 获取对应的平方数
List<Integer> squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList())
```

filter

filter 方法用于通过设置的条件过滤出元素。以下代码片段使用 filter 方法过滤出空字符串：

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
// 获取空字符串的数量
long count = strings.stream().filter(string -> string.isEmpty()).count();
```

limit

limit 方法用于获取指定数量的流。 以下代码片段使用 limit 方法打印出 10 条数据：

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

sorted

sorted 方法用于对流进行排序。以下代码片段使用 sorted 方法对输出的 10 个随机数进行排序：

```
Random random = new Random();
random.ints().limit(10).sorted().forEach(System.out::println);
```

并行（parallel）程序

parallelStream 是流并行处理程序的代替方法。以下实例我们使用 parallelStream 来输出空字符串数量：

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
// 获取空字符串的数量
int count = strings.parallelStream().filter(string -> string.isEmpty()).count();
```

我们可以很容易的在顺序运行和并行直接切换。

Collectors

Collectors 类实现了很多归约操作，例如将流转换成集合和聚合元素。Collectors 可用于返回列表或符串：

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList
));

System.out.println("筛选列表: " + filtered);
String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joini
g(", "));
System.out.println("合并字符串: " + mergedString);
```

统计

另外，一些产生统计结果的收集器也非常有用。它们主要用于int、double、long等基本类型上，它可以用来产生类似如下的统计结果。

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = numbers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("列表中最大的数 : " + stats.getMax());
System.out.println("列表中最小的数 : " + stats.getMin());
System.out.println("所有数之和 : " + stats.getSum());
System.out.println("平均数 : " + stats.getAverage());
```

其他

- Date Time API – 加强对日期与时间的处理。
- Optional 类 – Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。
- Nashorn, JavaScript 引擎 – Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM运行特定的javascript应用。

转自 https://blog.csdn.net/weixin_43395911/article/details/106356156