Java 8深入剖析与实战

讲师：张龙

版权所有 北京圣思园教育
iprogramming.cn

何为Lambda表达式

- **Lambda**：In programming languages such as Lisp, Python and Ruby lambda is an operator used to denote **anonymous functions** or **closures**, following the usage of lambda calculus

为何需要Lambda表达式

- 在Java中，我们无法将函数作为参数传递给一个方法，也无法声明返回一个函数的方法
- 在JavaScript中，函数参数是一个函数，返回值是另一个函数的情况是非常常见的；JavaScript是一门非常典型的函数式语言

Java匿名内部类示例

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button firstButton = (Button) findViewById(R.id.first);
        Button secondButton = (Button) findViewById(R.id.second);
        firstButton.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                        goToFirstActivity();
                }
        });
        secondButton.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                        goToSecondActivity();
                }
        });
}
```

Lambda表达式作用

- Lambda表达式为Java添加了缺失的函数式编程特性，使我们能将函数当做一等公民看待
- 在将函数作为一等公民的语言中，Lambda表达式的类型是函数。但在Java中，Lambda表达式是对象，他们必须依附于一类特别的对象类型——函数式接口(functional interface)

外部迭代

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

for (int number : numbers) {
    System.out.println(number);

}
```

内部迭代

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

numbers.forEach(new Consumer<Integer>() {
    public void accept(Integer value) {
        System.out.println(value);
    }
});
```

再进一步

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.forEach((Integer value) -> System.out.println(value));
```

更进一步

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

numbers.forEach(value -> System.out.println(value));
```

继续

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

numbers.forEach(System.out::println);
```

示例：排序

- 传统方式对字符串集合排序：

```
List<String> names = Arrays.asList("zhangsan", "lisi", "wangwu", "zhaoliu");
Collections.sort(names, new Comparator<String>() {
        @Override
        public int compare(String a, String b) {
            return b.compareTo(a);
        }
 });
System.out.println(names);
```

Lambda方式

```
List<String> names2 = Arrays.asList("zhangsan", "lisi", "wangwu", "zhaoliu");

Collections.sort(names2, (a, b) -> b.compareTo(a));

System.out.println(names2);
```

Java Lambda概要

- Java Lambda表达式是一种匿名函数；它是没有声明的方法，即没有访问修饰符、返回值声明和名字

Lambda表达式作用

- 传递行为，而不仅仅是值
  - 提升抽象层次
  - API重用性更好
  - 更加灵活

Java Lambda基本语法

- Java中的Lambda表达式基本语法
  - (argument) -> (body)
- 比如说
  - (arg1, arg2...) -> { body }
  - (type1 arg1, type2 arg2...) -> { body }

Java Lambda示例

- Lambda示例说明
  - (int a, int b) -> { return a + b; }
  - () -> System.out.println("Hello World");
  - (String s) -> { System.out.println(s); }
  - () -> 42
  - () -> { return 3.1415 };

Java Lambda结构

- 一个 Lambda 表达式可以有零个或多个参数
- 参数的类型既可以明确声明，也可以根据上下文来推断。例如：(int a)与(a)效果相同
- 所有参数需包含在圆括号内，参数之间用逗号相隔。例如：(a, b) 或 (int a, int b) 或 (String a, int b, float c)
- 空圆括号代表参数集为空。例如：() -> 42

Java Lambda结构

- 当只有一个参数，且其类型可推导时，圆括号（）可省略。例如：a -> return a*a
- Lambda 表达式的主体可包含零条或多条语句
- 如果 Lambda 表达式的主体只有一条语句，花括号{}可省略。匿名函数的返回类型与该主体表达式一致
- 如果 Lambda 表达式的主体包含一条以上语句，则表达式必须包含在花括号{}中（形成代码块）。匿名函数的返回类型与代码块的返回类型一致，若没有返回则为空

函数式接口

- 函数式接口是只包含一个抽象方法声明的接口
- java.lang.Runnable 就是一种函数式接口，在 Runnable 接口中只声明了一个方法 void run()
- 每个 Lambda 表达式都能隐式地赋值给函数式接口

FunctionalInterface

- java.lang.FunctionalInterface
  - 标识所声明的接口为函数式接口
  - 如果不满足函数式接口的要求，则编译器报错
  - 并非必须，但凡满足函数式接口条件的接口，编译器均将其看作是函数式接口，即便没有添加FunctionalInterface注解亦如此

何为传递行为?

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);

打印所有元素?

不打印任何一个元素?

打印所有偶数?

打印大于4的所有数字?

效率问题

- List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
- 问题：
  - 对于上述集合中的每一个元素，找出偶数、将其乘以2，然后打印出第一个大于5的元素值

直观做法

```
for (int number : numbers) {
    if (number % 2 == 0) {
        int n2 = number * 2;
        if (n2 > 5) {
            System.out.println(n2);
            break;
        }
    }
}
```

参见Test7.java

问题

- 逻辑不清晰
- 多个逻辑放在了一个方法中，不符合单一职责原则

改进

```
public boolean isEven(int number) {
    return number % 2 == 0;
}
public int doubleIt(int number) {
    return number * 2;
}
public boolean isGreaterThan5(int number) {
    return number > 5;
}
```

改进

```
List<Integer> l1 = new ArrayList<Integer>();
for (int n : numbers) {
    if (isEven(n)) l1.add(n);
}
List<Integer> l2 = new ArrayList<Integer>();
for (int n : l1) {
    l2.add(doubleIt(n));
}
List<Integer> l3 = new ArrayList<Integer>();
for (int n : l2) {
    if (isGreaterThan5(n)) l3.add(n);
}
System.out.println(l3.get(0));
```

参见Test8.java

问题

- 效率低下

流

- Collection提供了新的stream()方法
- 流不存储值，通过管道的方式获取值
- 本质是函数式的，对流的操作会生成一个结果，不过并不会修改底层的数据源，集合可以作为流的底层数据源
- 延迟查找，很多流操作（过滤、映射、排序等）都可以延迟实现

Optional

- 防止出现NullPointerException
- Google Guava等库提供了Optional的实现

```
Optional<SomeType> someValue = myTest();
if (someValue.isPresent()) {
    someValue.get().myMethod()
}
```

- 参见Test10.java

Stream

- Java 8 中的 Stream 是对集合对象功能的增强，它专注于对集合对象进行各种非常便利、高效的聚合操作，或者大批量数据操作
- Stream API 借助于Lambda 表达式，极大地提高了编程效率和程序可读性
- 提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势，使用 fork/join 并行方式来拆分任务和加速处理过程
- Stream 不是集合元素，它不是数据结构，并不保存数据，它是有关算法和计算的

Stream

- Stream更像一个高级版本的 Iterator。
- 原始版本的 Iterator，用户只能显式地一个一个遍历元素并对其执行某些操作；高级版本的 Stream，用户只要给出需要对其包含的元素执行什么操作，比如 “过滤掉长度大于 10 的字符串”、“获取每个字符串的首字母”等，Stream 会隐式地在内部进行遍历，并做出相应的数据转换
- Stream 就如同一个迭代器（Iterator），单向，不可往复，数据只能遍历一次

Stream

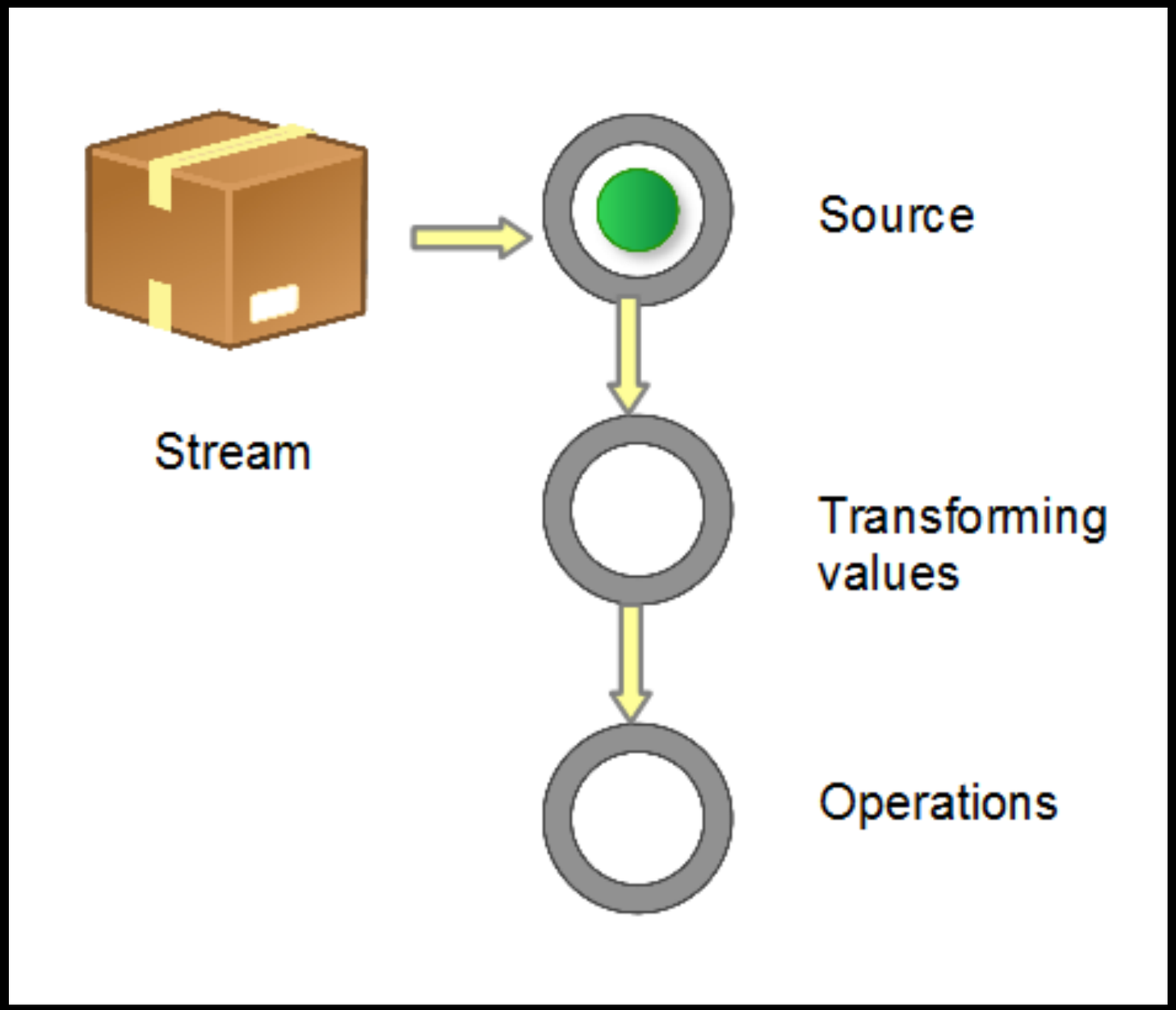
- 和迭代器又不同的是，Stream 可以并行化操作，迭代器只能命令式地、串行化操作
- 当使用串行方式去遍历时，每个 item 读完后再读下一个 item
- 使用并行去遍历时，数据会被分成多个段，其中每一个都在不同的线程中处理，然后将结果一起输出
- Stream 的并行操作依赖于 Java7 中引入的 Fork/Join 框架

Stream构成

- 获取一个数据源（source）→ 数据转换→执行操作获取想要的结果

- 每次转换原有 Stream 对象不改变，返回一个新的 Stream 对象（可以有多次转换），这就允许对其操作可以像链条一样排列，变成一个管道（Pipeline）

Stream



Stream源生成方式

- 从 Collection 和数组
  - Collection.stream()
  - Collection.parallelStream()
    - Arrays.stream(T array) or Stream.of()
- java.util.stream.IntStream.range()
- java.nio.file.Files.walk()
- ……

Stream操作类型

- Intermediate：一个流可以后面跟随零个或多个 intermediate 操作。其目的主要是打开流，做出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用，这类操作都是延迟的（lazy），就是说，仅仅调用到这类方法，并没有真正开始流的遍历

- Terminal：一个流只能有一个 terminal 操作，当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。Terminal 操作的执行，才会真正开始流的遍历，并且会生成一个结果

Stream效率

- 多个中间操作会导致循环集合多次么?

Stream使用

- 对 Stream 的使用就是实现一个 filter-map-reduce 的过程，最终产生一个结果
- 参见Stream1.java

Stream使用

- 对于原生数据类型，提供了IntStream、LongStream与DoubleStream
- 当然我们也可以用 Stream<Integer>、Stream<Long>、Stream<Double>，但是 boxing 和 unboxing 会很耗时，所以特别为这三种基本数值型提供了对应的 Stream

原生Stream的构造

- 参见Stream2.java

将Stream转换为其他类型

- 参见Stream3.java

Stream操作

- Intermediate
  - map (mapToInt, flatMap 等)、 filter、 distinct、 sorted、 peek、 limit、 skip、 parallel、 sequential、 unordered
- Terminal
  - forEach、 forEachOrdered、 toArray、 reduce、 collect、 min、 max、 count、 anyMatch、 allMatch、 noneMatch、 findFirst、 findAny、 iterator

Stream操作

- 参见Stream4.java
- 参见Stream5.java

并行Stream

- stream()
- parallelStream()
- 参见Stream6.java

默认方法

- Java 8中可以在接口中添加方法实现，只需在方法声明前加上default关键字即可
- 参见InterfaceTest.java

感谢对圣思园的支持

版权所有 北京圣思园教育
iprogramming.cn