day13_Stream流、方法引用

一.Stream 流

在 Java 8中，得益于 Lambda 表达式所带来的函数式编程，引入了一个全新的 Stream 流的概念，用于解决已有集合类库既有的弊端。下面我们通过一个案例来介绍 Stream 流的优点和用法。

(1).使用传统的方式遍历集合，对集合中的数据进行过滤

几乎所有的集合(如 Collection 接口、Map 接口等)都支持直接或间接的遍历操作。当我们需要对集合中的元素进行操作的时候，除了必需的添加、删除、获取等操作外，最典型的就是集合遍历。集合的遍历操作一般使用循环来实现，如下例：使用 for 循环来遍历集合，对集合中的数据进行条件过滤。

```
public class DemoMain {
    public static void main(String[] args) {

        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");

        // 对list集合中的元素进行过滤，只要以张开头的元素
        List<String> zhangList = new ArrayList<>();
        for (String name : list) {
            if (name.startsWith("张")) {
                zhangList.add(name);
            }
        }

        // 对zhangList集合中的元素进行过滤，只要姓名长度为3的人
        List<String> shortList = new ArrayList<>();
        for (String name : zhangList) {
            if (name.length() == 3) {
                shortList.add(name);
            }
        }

        // 遍历shortList集合并输出
        for (String name : shortList) {
            System.out.println(name);
        }

    }
}
```

循环与遍历的关系：

Java 8中引入的 Lambda 表达式让我们可以更加专注于做什么(What)，而不是怎么做(How)，这点此前已经结合匿名内部类进行了对比说明。现在我们仔细体会一下上例代码，可以发现：

- for 循环的语法就是“怎么做”
- for 循环的循环体才是“做什么”

为什么要使用循环？因为要对集合进行遍历。但循环是遍历集合的唯一方式吗？遍历是指对每一个元素逐一进行处理，而并不是从第一个到最后一个顺次处理的循环。前者是目的，后者是方式，因此我们只是想实现集合遍历这一目的(做什么)，才使用了循环这一方式来实现遍历(怎么做)，当然实现集合遍历并不只有循环这一种方式。

循环遍历的弊端：

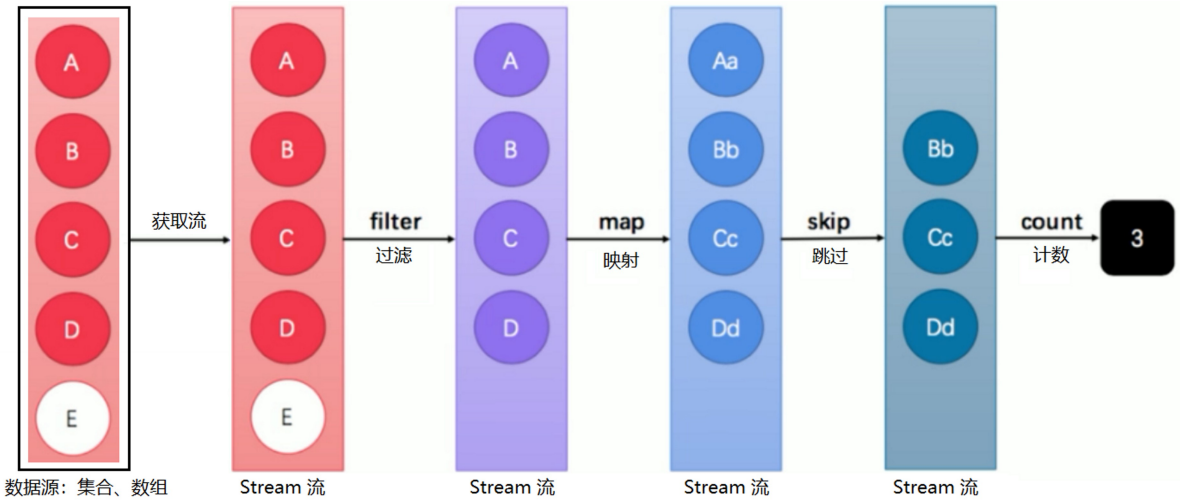
在上述代码中，每当我们需要对集合中的元素进行操作的时候，总是需要进行循环、循环、再循环。这是理所当然的么？并不是，循环只是做事情的方式，而不是目的。另一方面，使用线性循环就意味着只能遍历一次，如果希望再次遍历，只能再使用另一个循环从头开始。那 Lambda 的衍生物 Stream 流能给我们带来怎样更加优雅的写法呢?

(2).使用 Stream 流的方式遍历集合，对集合中的数据进行过滤

```
public class DemoMain {
    public static void main(String[] args) {

        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");

        list.stream() // 获取流
                .filter(name -> name.startsWith("张"))      // 只要以张开头的元素
                .filter(name -> name.length() == 3)         // 只要姓名长度为3的人
                .forEach(name -> System.out.println(name)); // 输出过滤后的结果

    }
}
```

直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：获取流、过滤姓张、过滤长度为3、逐一打印。代码中并没有体现使用线性循环或是其他任何算法进行遍历，我们真正要做的事情内容被更好地体现在代码中。

Stream 流是一个来自数据源的元素队列，其中元素是特定类型的对象，形成一个队列，数据源可以是集合、数组等。Stream 流并不是集合、也不是数据结构，其本身不存储任何元素或其地址值，而是用来对"来自数据源的元素"进行按需计算(数据处理)。



上图就展示了如何使用 Stream 流对集合元素进行一系列的处理(多步操作)：首先根据数据源获取到一个 Stream 流，然后使用该 Stream 流调用 filter、map、skip、count 等方法，对元素进行过滤、映射、跳过、计数等多步操作。图中的每一个方框都是一个 Stream 流，即调用指定的方法，就可以从一个流模型转换为另一个流模型，即 filter、map、skip 等方法的返回值是一个新的、元素经过指定方法处理后的 Stream 流，count 方法是最终用来计数的方法，返回值是一个整数。Stream 流对元素的操作具有以下特征：

- Pipelining：中间操作都会返回 Stream 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。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 或者增强 for 的方式，显式的在集合外部进行迭代，Stream 流提供了内部迭代的方式，流对象可以直接调用遍历方法 foreach。
- 使用 Stream 流对集合元素进行处理时，集合元素并没有真正被处理。由于 Stream 流不能存储数据，所以无论 Stream 流怎么操作数据，集合本身并不会发生改变。

使用一个 Stream 流的时候，通常包括三个基本步骤：获取一个数据源 → 数据转换 → 执行操作获取想要的结果，每次数据转换时原有的 Stream 流对象不改变，返回一个新的 Stream 流对象，这就允许"对数据的转换操作"可以像链条一样排列，形成一个管道。

1.获取 Stream 流

java.util.stream.Stream<T> 接口是 Java 8 新加入一个 Stream 流接口(这并不是一个函数式接口)，Stream 流对象就是该接口的一个实现类对象。获取一个 Stream 流对象有以下两种常用的方式：

- 所有的 Collection 集合都可以通过 stream 默认方法获取流
- Stream 接口的静态方法 of 可以获取数组对应的流

2.常用方法

Stream 流的操作很丰富，这里介绍一些常用的 API。这些方法可以被分成两种类型：

- 延迟方法：返回值类型仍然是 Stream 接口类型，因此支持链式调用。(除了终结方法外，其余方法均为延迟方法)
- 终结方法：返回值类型不再是 Stream 接口类型，因此不再支持链式调用，用来终结一个 Stream 流的操作。(forEach、count 方法为终结方法)

(1).forEach 方法

(2).filter 方法

(3).map 方法

(4).count 方法

(5).limit 方法

(6).skip 方法

(7).concat 方法

#.注意事项：
①.当 Stream 流对象采用"链式调用的方式"来调用上述方法时，得益于 Lambda 表达式延迟执行的特性，只有当终结方法执行的时候，之前的延迟方法才会执行，因此被称为延迟方法。

```
public class DemoMain {
    public static void main(String[] args) {

        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张三丰");

        // 只要终结方法forEach不执行，前面的两个filter方法就不会执行
        list.stream()
                .filter(name -> name.startsWith("张"))
                .filter(name -> name.length() == 3)
                .forEach(name -> System.out.println(name));

    }
}
```

②.Stream 流属于管道流，只能被使用一次，即一个 Stream 流调用完毕方法后，数据就会流转到下一个 Stream 流上，而此时前一个 Stream 流对象已经不能关闭，不能再使用了，如果再次使用则会报出异常。

```java
public class DemoMain {
    public static void main(String[] args) {

        // 获取一个Stream流
        Stream<String> stream01 = Stream.of("张三丰", "张翠山", "赵敏", "张无忌");

        // 使用filter方法对Stream流中的数据进行过滤
        Stream<String> stream02 = stream01.filter(name -> name.startsWith("张"));

        // 遍历stream02流
        stream02.forEach(name -> System.out.println(name));

        // 再次使用stream01流，则会报出异常IllegalStateException
        stream01.forEach(name -> System.out.println(name));
    }
}
```

3.练习：分别使用传统方式、Stream 流的方式来处理集合元素

二.方法引用

在使用 Lambda 表达式的时候，我们实际上传递进去的一段方法体代码就是一种解决方案：拿什么参数做什么操作。如果我们在 Lambda 表达式中所指定的操作方案，已经有一个"已存在的方法"实现了这个功能，那么我们就可以直接传递这个"已存在的方法"给函数式接口，来代替传递一个 Lambda 表达式，而没有必要在 Lambda 表达式中再重写方法体代码。这种使用已存在的方法来代替 Lambda 表达式的方式，就叫做方法引用。

```java
// 定义一个用来打印的函数式接口
@FunctionalInterface
public interface Printable {

    // 定义用来打印字符串的抽象方法
    void print(String s);

}
public class DemoMain {
    public static void main(String[] args) {

        // 调用printString方法，传递一个Lambda表达式
        printString( s: "HelloWorld", s -> System.out.println(s));

        // 调用printString方法，传递一个方法引用，即：使用方法引用来代替Lambda表达式
        printString( s: "HelloWorld", System.out::println);

    }

    // 定义一个方法printString，参数传递一个字符串和Printable接口，用来打印输出一个字符串
    private static void printString(String s, Printable printable) {
        printable.print(s);
    }

}
```

上述代码中：传递的 Lambda 表达式的方法体就是调用 println 方法来打印输出字符串，即已经有一个"已存在的 println 方法"实现了这个功能，所以我们可以直接将 System.out 对象的 println 方法传递给函数式接口，来代替传递这个 Lambda 表达式。

双冒号 : : 的写法是方法引用的语法，称为引用运算符，而它所在的表达式被称为方法引用表达式，如果 Lambda 要表达的函数方案已经存在于某个方法的实现中，那么则可以通过双冒号来引用该方法作为 Lambda 的替代者。在上述代码中：使用 System.out 对象来引用对象中的 println 方法，来代替 Lambda 表达式并传递给函数式接口。所以方法引用就是复用了已有方案，来代替 Lambda 表达式的语法，更加简洁。

#.注意事项:

(1).Lambda 表达式中传递的参数，一定是方法引用中的那个方法可以接收的类型，否则会抛出异常

(2).在上述案例中，如果使用 Lambda 表达式，那么根据"可推导就可省略"的原则，无需指定参数 s 的类型为 String，也无需指定 println 方法是哪一个重载形式，它们都将被自动推导出来。而如果使用方法引用，这些内容同样也可以根据上下文进行推导。如下例：把上述代码的函数式接口改为 int 类型的参数，方法引用会结合上下文自动推导出 println 方法唯一对应的匹配重载，所以方法引用无需变化。

```java
// 定义一个用来打印的函数式接口
@FunctionalInterface
public interface PrintableInteger {

    // 定义用来打印整数的抽象方法
    void print(int i);

}

public class DemoMain {
    public static void main(String[] args) {

        // 调用printString方法，传递一个Lambda表达式
        printString( i: 123, i -> System.out.println(i)); // 自动匹配到 println(int x)方法

        // 调用printString方法，传递一个方法引用，即：使用方法引用来代替Lambda表达式
        printString( i: 123, System.out::println); // 自动匹配到 println(int x)方法

    }

    // 定义一个方法printString，参数传递一个整数和PrintableInteger接口，用来打印输出一个整数
    private static void printString(int i, PrintableInteger printableInteger) {
        printableInteger.print(i);
    }

}
```

1.通过对象名引用成员方法

2.通过类名引用静态方法

3.通过 super、this 引用重名的父、子类成员方法

4.类的构造器引用

5.数组的构造器引用