Java 8 Stream API编程基础

北京理工大学计算机学院
金旭亮

主要内容

- Stream API概述
- 流的创建方法
- 常用类型
- 流的基本操作

1、Stream API概述

概述

- Java 8中新特性中，最引人注目的有两项：一是Lambda表达式，二是Stream API。
- Stream API使用了很多Lambda表达式的特性，主要用于集合操作。
- 注意Stream API中的“Stream”，与文件操作中的“Stream”含义是不一样的，Stream API中的Stream，可以看成一种“特殊的集合”，我们可以向这个集合施加一系列的“操作（Operation）”。

引例

- 假设我们有一个学生对象的集合，希望知道所有来自于北京的学生人数。传统的编程方式是这样的：

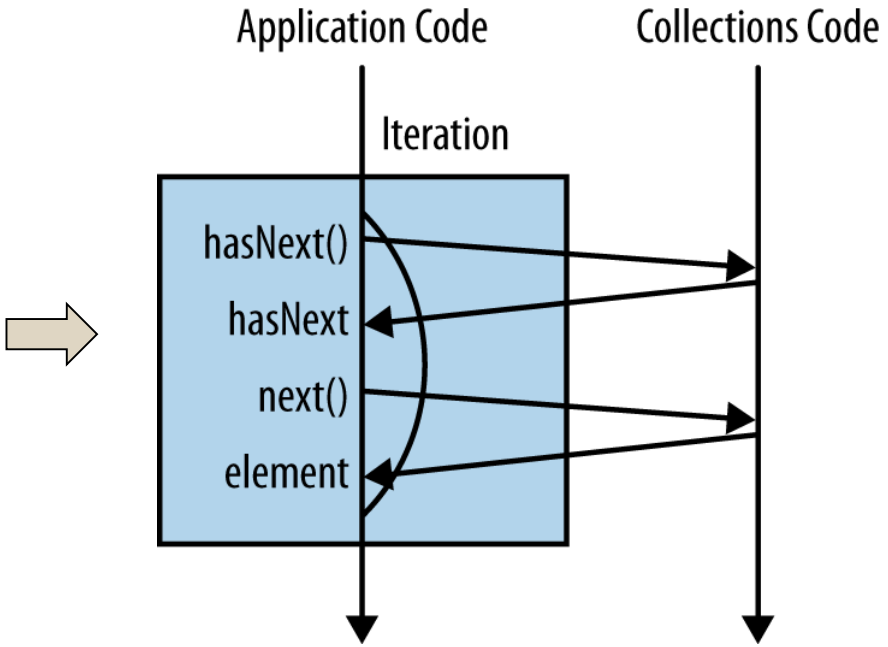
```
//传统的使用for循环的编程方式
static void filterWithForLoop(){
    int count=0;
    List<Student> students=Student.getStudents();
    for(Student stu:students){
        if(stu.getFrom().equals("北京"))
            count++;
    }
    System.out.println("来自于北京有学生有："+count+"名");

}
```

示例：WhatIsStream.java

传统编程模式的问题

- 如果搜索条件改变，必须重写代码
- 如果希望并行处理，则必须重写每个循环
- 要想了解循环干了什么，必须仔细地阅读所有代码

Java的ForEach循环，本质上是通过iterator实现的，所有处理工作，在同一线程，同一循环内部完成。



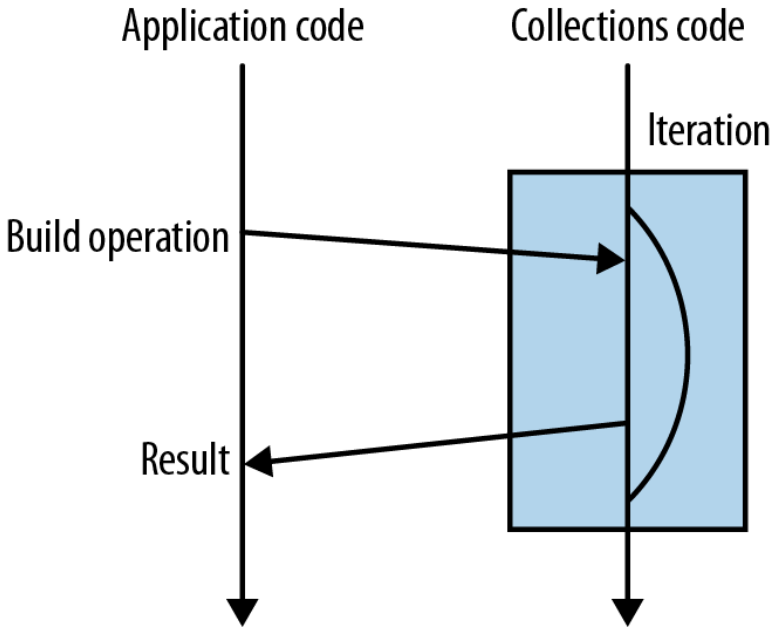
如果使用Stream API……

```
//使用Stream API的编程方式
static void filterWithStreamAPI(){
    long count=Student.getStudents()
            .stream()
            .filter(stu->stu.getFrom().equals("北京"))
            .count();
    System.out.println("来自于北京有学生有："+count+"名");
}
```

- 一句完成了所有工作：遍历→过滤→统计
- 上述代码中，stream()方法返回一个“流”，filter和count都是对这个流所施加的“操作”，这种操作可以级联，最后一个才是负责真正启动执行这一连串处理工作的操作（称为“Terminal Operation”，本例中是count），中间的称为“Intermediate Opearation”，它们是被动的，不会引发操作的执行。
- 下面我们来看看Stream API的执行示意图

Stream API将集合操作“外置”

使用Stream API，调用者只管设定好级联的操作并启动之，所有操作都是由“Stream”完成的，调用者最后直接拿结果就行了。



这种编程方式，只需告诉计算机“Do what”，而无需告诉计算机“How to Do”，具体如何完成由Stream API自己决定。

并行处理

```
// 使用Stream API的并行编程方式
static void filterWithParallelStreamAPI() {
    System.out.println("线程："+Thread.currentThread().getId()
            +"启动处理工作");
    long count = Student
            .getStudents()
            .parallelStream()
            .peek(stu -> {
                System.out.println("线程" + Thread.currentThread().getId()
                        + "正在处理" + stu);
            }).filter(stu -> stu.getFrom().equals("北京")).count();
    System.out.println("线程："+Thread.currentThread().getId()+
            "显示处理结果：来自于北京有学生有" + count + "名");
}
```

```
线程：1启动处理工作
线程10正在处理(王五,北京)
线程10正在处理(赵六,重庆)
线程12正在处理(朱十,上海)
线程12正在处理(赵九,北京)
线程1正在处理(张八,上海)
线程11正在处理(李四,上海)
线程12正在处理(韩七,南京)
线程10正在处理(张三,北京)
线程：1显示处理结果：来自于北京有学生有3名
```

- **我们可以很方便地以多线程方式并行处理集合元素……**

不需要与Thread直接打交道！

并行循环

- 当采用并行方式执行遍历时，元素的访问顺序将变得“混乱”。

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

    System.out.println("创建字符串集合");
    List<String> strings = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        strings.add("Item " + i);
    }
    strings.stream()
        .parallel()
        .forEach(str -> System.out.println(str));
}
```

```
创建字符串集合
Item 6
Item 5
Item 8
Item 9
Item 7
Item 2
Item 1
Item 3
Item 4
Item 0
```

参看示例ParalllelStreams.java

理解“Stream”

- Stream API中的“Stream（流）”，可以看成是一组数据的集合，我们可以对它施加一系列的数据操作，这些操作可以是顺序进行的，也可以是并行执行的。
- Stream看上去是集合，但实际上不是集合，为什么呢?
- 下面列出一下Stream与普通集合（比如ArrayList）的不同之处

Stream的特性-1

- 普通的集合，关注的是数据的存储方式，而Stream，关注的是施加于这些集合元素的处理工作（称为“Operation（操作）”）。
- Stream通常从普通的集合中“抽取”数据，但也可以从其它数据源中提取数据。
- Stream只“抽取”数据，它从不会修改底层的数据源。简言之：Stream Operation是只读操作!
- 只要底层数据源允许，流中包容的元素数目是不受限制的。

Stream的特性-2

- 使用Stream，开发者可以很方便地让代码并行执行而无需显式编写多线程代码。
- 基于Stream的代码，具有“函数式编程（functional programming）”风格。
- Stream只能使用一次，如果希望再次执行这些操作，必须创建一个新的Stream。

流对象只能用一次!

```java
//试图重用一个Stream，引发异常
private static void StreamCannotBeReuse() {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    Stream<Integer> stream=numbers.stream();
    System.out.println("numbers集合中有元素"+stream.count()+"个");
    //以下这句将引发java.lang.IllegalStateException
    System.out.println("平均值为："+
            stream.mapToInt(num->num.intValue()).average());
}
```


```java
//只有被重建之后，才能再次执行操作
private static void StreamMustBeRecreated() {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    Stream<Integer> stream=numbers.stream();
    System.out.println("numbers集合中有元素"+stream.count()+"个");
    //重建一个流对象
    stream=numbers.stream();
    System.out.println("平均值为："
            +stream.mapToInt(num->num.intValue()).average());
}
```

推荐的“重用流”的编程方式

```
//通过构建一个Supplier对象，重复创建特定的流
private static void StreamRecreatedViaSupplier() {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    Supplier<IntStream> streamSupplier=
            ()->numbers.stream().mapToInt(num->num.intValue());
    //使用Supplier接口所定义的get()方法，获取流对象的新实例
    System.out.println("numbers集合中有元素"
            +streamSupplier.get().count()+"个");
    System.out.println("平均值为："
            +streamSupplier.get().average());
}
```

- **当需要反复执行特定流的不同操作系列时，可以构建一个Supplier对象，通过它获取新的流实例。**

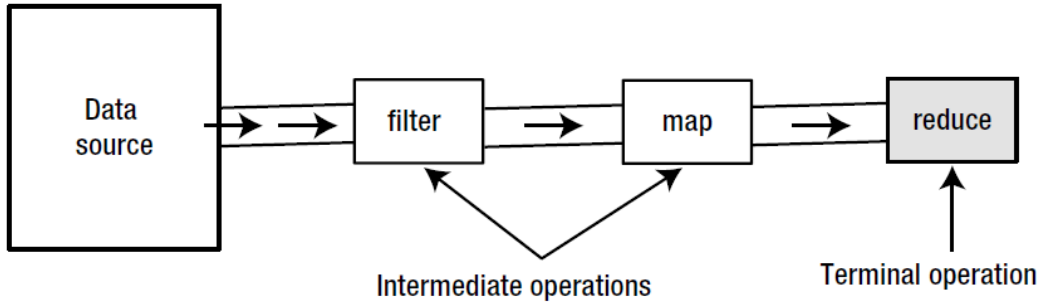
示例代码：ReUseStream.java

理解“流处理管线”

- 我们来看一个例子，求集合中所有奇数的平方和（示例：SumOfOddNumbers）

```
public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    int sum = numbers.stream()
            .filter(n -> n % 2 == 1)
            .map(n -> n * n)
            .reduce(0, Integer::sum);
    System.out.println("Result:"+sum);
}
```

- 上述代码实际上构建一个了数据处理管线，如下图所示：



构成流处理管线的两种操作类型

- 流处理管线由“中间操作”和“终止操作”所构成。数据流入“中间操作”，中间操作可以对这些数据进行特定的加工和处理，再将其转发给下一个操作，就象工厂中的流水线一样。
- 终止操作完成以下任务：

1. 启动执行整个流操作系列
2. 向调用者返回最终结果
3. 操作结束后，销毁流对象

流处理管线示例：SumOfOddNumbers.java

```
创建流

处理元素 1，执行filter
处理filter结果元素 1，执行map
处理map结果元素 1，执行reduce

处理元素 2，执行filter

处理元素 3，执行filter
处理filter结果元素 3，执行map
处理map结果元素 9，执行reduce

处理元素 4，执行filter

处理元素 5，执行filter
处理filter结果元素 5，执行map
处理map结果元素 25，执行reduce

流处理结束，流处理管线结果为：35
```

- **示例分析：**

1. **集合中的元素为：1,2,3,4,5**
2. **可以看到，只有奇数走完了整个管线。**
3. **filter操作负责“过滤”出奇数，map操作负责计算出奇数的平方和**
4. **reduce操作调用Integer的sum方法不断累加每个奇数的平方和，生成最终结果，然后销毁整个管线。**

2、流的创建方法

概述

- 为了支持级联调用，所有流的“中间操作”都会返回一个流对象
- 但我们总需有一个“初始”的流对象，以便基于它开始构建一个流处理管线。
- 下面的PPT中介绍如何从原始数据源中创建这个“初始”的流对象。

基于集合创建Stream

- 这可能是最常用的流创建方法了。
- 所有集合对象都实现的Collection接口定义了一个Stream()/parallelStream()方法，可以通过它来创建流对象：

```
//创建一个集合对象
Set<String> names = new HashSet<>();
names.add("Ken");
names.add("jeff");
// 基于集合对象创建流
Stream<String> sequentialStream = names.stream();
//基于集合对象创建并行流（ parallel stream）
Stream<String> parallelStream = names.parallelStream();
```

使用Stream.Of创建流

- Stream接口定义了一个of方法，可以用于创建流

```
//使用可变参数创建流
Stream<String> stream = Stream.of("Ken", "Jeff", "Chris", "Ellen");
```

```
//基于数组创建流
String[] names = {"Ken", "Jeff", "Chris", "Ellen"};
Stream<String> stream = Stream.of(names);
```

Stream.Builder<T>接口

- 使用Stream.Builder<T>接口也能创建流

```
Stream<String> stream = Stream.<String>builder()
                                .add("Ken")
                                .add("Jeff")
                                .add("Chris")
                                .add("Ellen")
                                .build();
```

基于数组构建流

- JDK中的Arrays类提供了直接从对象数组中创建流的静态方法stream()，以字符串数组为例：

```
//从字符串数组中创建一个字符串流
Stream<String> names = Arrays.stream(
        new String[] {"Ken", "Jeff"});
```

构建数值型流-1

- 针对int、long、double这三种原始数值类型，java 8提供了单独的流类型：IntStream、LongStream和DoubleStream
- 例如：

```
//使用可变参数构建流
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
```

```
//使用数组构建流
int[] values={1,2,3,4};
IntStream stream = Arrays.stream(values);
```

生成指定范围内的整数流

- **以IntStream为例介绍，LongStream的用法也是类似的。**

```
//生成的流包容 [0,100) 区间的所有整数
IntStream zeroToNinetyNine = IntStream.range(0, 100);

//生成的流包容 [0,100]区间的所有整数
IntStream zeroToHundred = IntStream.rangeClosed(0, 100);
```

构建一个不包容任何元素的流

- 在有些情况下，需要构建一个“空”的流，调用Stream.empty()方法可以达到这个目的：

```
//构建一个空的字符串流
Stream<String> stream = Stream.empty();
//构建一个空的整数流
IntStream numbers = IntStream.empty();
```

基于函数构建流——元素工厂

- 我们可以定义一个函数，这个函数可以帮助我们构建出流的元素，这个函数可以看成是生成流元素的“元素工厂”，然后使用这个工厂生产出来的“元素”构建出流。

```
//使用流元素工厂构建流
private static void generateExample() {
    //整数生产“流水线”，每次调用，生成一个[0,100)区间内的整数
    Supplier<Integer> intFactory = () -> (int) (Math.random() * 100);
    //调用整数元素“工厂”生产出10个“整数” 产品
    Stream<Integer> stream = Stream.generate(intFactory).limit(10);
    //输出所有元素
    stream.forEach(System.out::println);
}
```

示例：StreamElementFactory.java

基于函数构建流——迭代法

- 有些流的元素需要依据它的前一个元素才能确定，对于这种情况，可以使用Stream.iterate()方法构建流对象：

```
//使用迭代法构建流
private static void iterateExample(){
    //起始值为1
    int seed=1;
    //使用递推公式：a[n]=a[n-1]+2生成奇数
    UnaryOperator<Integer> intFactory=n->n+2;
    //生成包容10个奇数的流
    Stream<Integer> stream=Stream.iterate(seed, intFactory).limit(10);
    //输出所有元素
    stream.forEach(System.out::println);
}
```

动手动脑

- 如何生成一个常量流?
- 比如生成一个流，其中的元素值全部都为100。

无穷流

- 使用Stream. generate和Stream.iterate方法构建的流，可以拥有“无数个元素”，称为“无穷流（infinite stream）”。
- 当然我们不能直接使用这个流，通常会使用limit()方法构建一个仅包容“有限元素”的流。

生成随机数值流的简便方法

- 鉴于程序中经常需要生成随机的整数和浮点数，JDK8中为Random类添加了ints()/longs()/doubles()方法创建相应的数值流。

```
//生成[0,100)区间中的整数
new Random().ints(0,100)
            .limit(5)
            .forEach(System.out::println);

//生成[100,200)
new Random().doubles(100,200)
            .limit(5)
            .forEach(System.out::println);
```

基于文件和文件夹创建流

- Java 8中，我们可以基于一个文本文件中的所有行构建流，也可以基于一个文件夹中的所有子文件夹构建流。
- 参看示例：IOStreamDemo.java

```
public static void readFileContents(String filePath) {
    Path path = Paths.get(filePath);
    if (!Files.exists(path)) {
        System.out.println("The file " + path.toAbsolutePath()
                + " does not exist.");
        return;
    }
    try (Stream<String> lines = Files.lines(path)) {
        // 读入文件中的所有行并输出
        lines.forEach(System.out::println);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

动手动脑

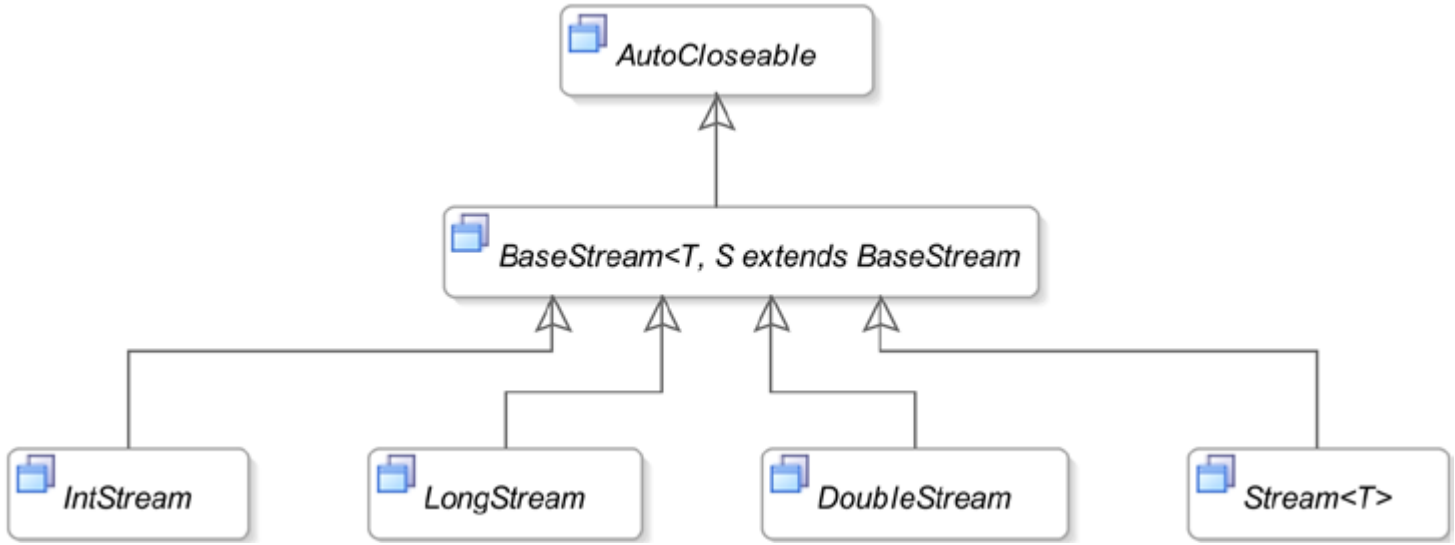
- 创建一个生成素数的流，然后使用它找出[100,10000]内的素数
- 先使用标准流执行它，再使用并行流执行它，比对这两种方式的执行效果

生成素数的流，可以参看示例PrimeUtil.java

流相关的类与函数式接口

JDK8中的Stream相关的类型

- 与Stream API相关的流类型继承树如下所示：



讨厌的NullPointerException

- 由于Stream API中经常需要“级联”多个操作，因此，如果中间某个操作遇到null引用时，如何处理它是个麻烦事。
- 默认情况下，如果针对null引用调用方法，JVM会抛出NullPointerException，如果不做妥善处理，可能会“打断”流处理管线。
- 为了解决这个问题，Java8专门引入了一个Optional<T>类。

Optional类型

- Optional对象是一个“可能为null”的对象。
- 所有那些有可能返回一个null引用的操作，都应该返回一个Optional对象。
- Optional对象定义了一个isPresent()方法用于确定它是否包容一个有效的值。如果值有效，调用它的get()方法提取出这个值。
- 如果isPresent()返回false，代码又尝试调用它的get()方法，JVM会抛出一个NoSuchElementException 。

Optional示例代码

- 有不少Stream API操作返回Optional对象，以下是一个示例：

```
String[] words={"this","is","a","test"};
Optional<String> largest = Arrays.stream(words)
                .max(String::compareToIgnoreCase);
if (largest.isPresent())
                System.out.println("largest: " + largest.get());
}
```

```
// Create an Optional for the string "Hello"
Optional<String> str = Optional.of("Hello");
//如果有数据，就打印它！
str.ifPresent(value -> System.out.println("Optional contains " + value));
```

构建Optional对象-1

- 最简单的构建Optional对象的方法是使用其of()方法，另一个empty()方法用于构建一个”空的”Optional对象。

```
//截取指定长度的字符串
public static Optional<String> processString(String string, int length) {
    if (string == null || length <= 0 ||string.length()<length) {
        return Optional.empty();
    }
    return Optional.of(string.substring(0, length));
}
```

示例：CreateOptionalDemo.java

构建Optional对象-2

- 另一个有用的方法是 Optional.ofNullable(obj)：
- 当obj==null时，此方法返回Optional.empty()，否则，返回Optional.Of(obj)

针对原始数据类型的Optional类型

- 针对原始数据类型，JDK提供了OptionalInt, OptionalLong和OptionalDouble三个类，拥有getAsXXX()的方法
- 下面以OptionalInt为例介绍其用法：

```
// Create an empty OptionalInt
OptionalInt empty = OptionalInt.empty();
// Use an OptionaInt to store 287
OptionalInt number = OptionalInt.of(287);
if(number.isPresent()){
      int value = number.getAsInt();
      System.out.println("Number is " + value);
}
else {
    System.out.println("Number is absent.");
}
```

Optional综合示例

- OptionalTest.java展示了Optional的一些用法，可供参考。

Stream API中所使用的函数式接口

Functional Interface	Parameter Types	Return Type	Description
`Supplier<T>`	None	`T`	Supplies a value of type `T`
`Consumer<T>`	`T`	`void`	Consumes a value of type `T`
`BiConsumer<T, U>`	`T, U`	`void`	Consumes values of types `T` and `U`
`Predicate<T>`	`T`	`boolean`	A Boolean-valued function

`ToIntFunction<T>`
`ToLongFunction<T>`
`ToDoubleFunction<T>`
	`T`	`int`
		`long`
		`double`
			An `int`-, `long`-, or `double`-valued function

`IntFunction<R>`
`LongFunction<R>`
`DoubleFunction<R>`
	`int`
	`long`
	`double`
		`R`	A function with argument of type `int`, `long`, or `double`

`Function<T, R>`	`T`	`R`	A function with argument of type `T`
`BiFunction<T, U, R>`	`T, U`	`R`	A function with arguments of types `T` and `U`
`UnaryOperator<T>`	`T`	`T`	A unary operator on the type `T`
`BinaryOperator<T>`	`T, T`	`T`	A binary operator on the type `T`

动手动脑

- 前页PPT所述之函数式接口，在Stream API的各个方法中频繁出现，通常在开发中，使用Lambda表达式或方法引用作为参数调用这些方法。
- 理解这些函数式接口的含义和用法是使用Stream API的前提。
- 这个任务留为课后作业。

流的基本操作

概述

- Stream API中定义了诸多的操作，能完成各种常见的数据处理工作。
- 了解并熟悉这些基本操作，需要花费一些时间，有些操作理解起来还会比较费劲，但这个时间是必须要花的，否则，你很难在开发中用好它们。
- 强调一下：掌握这些基本操作，要求扎实掌握Java 8的Lambda表达式特性，并且熟悉相关的函数式接口

常用的Stream API操作-1

操作名	说明
Distinct	Returns a stream consisting of the distinct elements of this stream. Elements e1 and e2 are considered equal if e1.equals(e2) returns true.
filter	Returns a stream consisting of the elements of this stream that match the specified predicate.
flatMap	Returns a stream consisting of the results of applying the specified function to the elements of this stream. The function produces a stream for each input element and the output streams are flattened. Performs one-to-many mapping.
limit	Returns a stream consisting of the elements of this stream, truncated to be nolonger than the specified size.

常用的Stream API操作-2

操作名	说明
map	Returns a stream consisting of the results of applying the specified function to the elements of this stream. Performs one-to-one mapping.
peek	Returns a stream whose elements consist of this stream. It applies the specified action as it consumes elements of this stream. It is mainly used for debugging purposes.
skip	Discards the first n elements of the stream and returns the remaining stream. If this stream contains fewer than n elements, an empty stream is returned.
sorted	Returns a stream consisting of the elements of this stream, sorted according to natural order or the specified Comparator. For an ordered stream, the sort is stable.

常用的Stream API操作-3

操作名	说明
allMatch	Returns true if all elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
anyMatch	Returns true if any element in the stream matches the specified predicate, false otherwise. Returns false if the stream is empty.
findAny	Returns any element from the stream. An empty Optional object is for an empty stream.
findFirst	Returns the first element of the stream. For an ordered stream, it returns the first element in the encounter order; for an unordered stream, it returns any element.

常用的Stream API操作-4

操作名	说明
noneMatch	Returns true if no elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
forEach	Applies an action for each element in the stream.
reduce	Applies a reduction operation to computes a single value from the stream.

常用的Stream API操作-1

操作名	说明
Distinct	Returns a stream consisting of the distinct elements of this stream. Elements e1 and e2 are considered equal if e1.equals(e2) returns true.
filter	Returns a stream consisting of the elements of this stream that match the specified predicate.
flatMap	Returns a stream consisting of the results of applying the specified function to the elements of this stream. The function produces a stream for each input element and the output streams are flattened. Performs one-to-many mapping.
limit	Returns a stream consisting of the elements of this stream, truncated to be no longer than the specified size.

综合示例项目StreamOptDemo

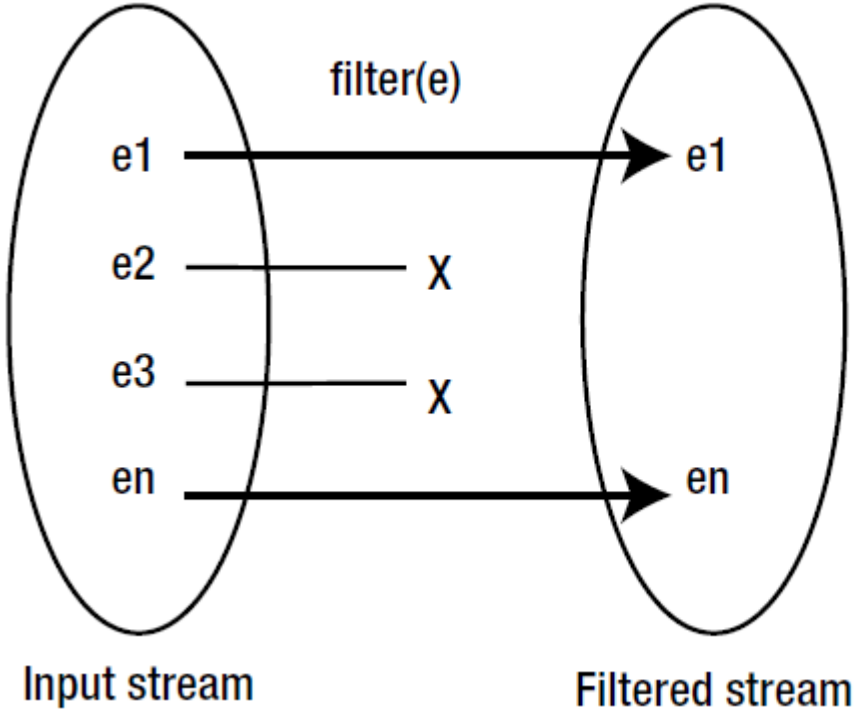
- 在此示例项目中，定义了一个Person类，然后针对一些常用的Stream API操作设计了一些测试代码，展示其基本用法
- 请同学们在PPT的引导下仔细阅读这些示例，掌握常用的Stream API操作的使用方法

遍历和跟踪

- 使用foreach()方法
- 参看ForEachTest.java
- 使用peek()方法跟踪当前处理的元素，这个方法多用于调试。
- 参看PeekTest.java

过滤

- 使用filter()方法实现过滤。
- 参看FilterTest.java



排序

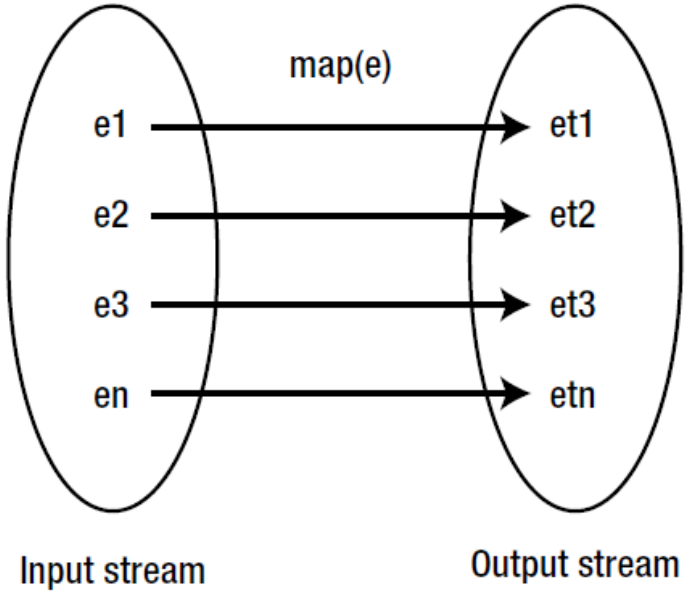
- 使用sorted()方法进行排序

```
public static void main(String[] args) {
    List<String> words=new ArrayList<>();
    words.add("apple");
    words.add("oranage");
    words.add("method ");
    words.add("reversed");
    Stream<String> longestFirst =
            words.stream().sorted(
                    Comparator.comparing(String::length));
    longestFirst.forEach(System.out::println);
}
```

示例：SortDemo.java

转换

- 使用map（）方法把一种类型的数据转换为另一种类型的数据，或者是完成特定的处理。



如果生成的结果是“元素是流的流（stream of stream）”，则需要使用flatMap()将形成上下级关系的两个流“合成为一个”。

参看MapDemo.java

消除重复元素

- 使用distinct（）方法

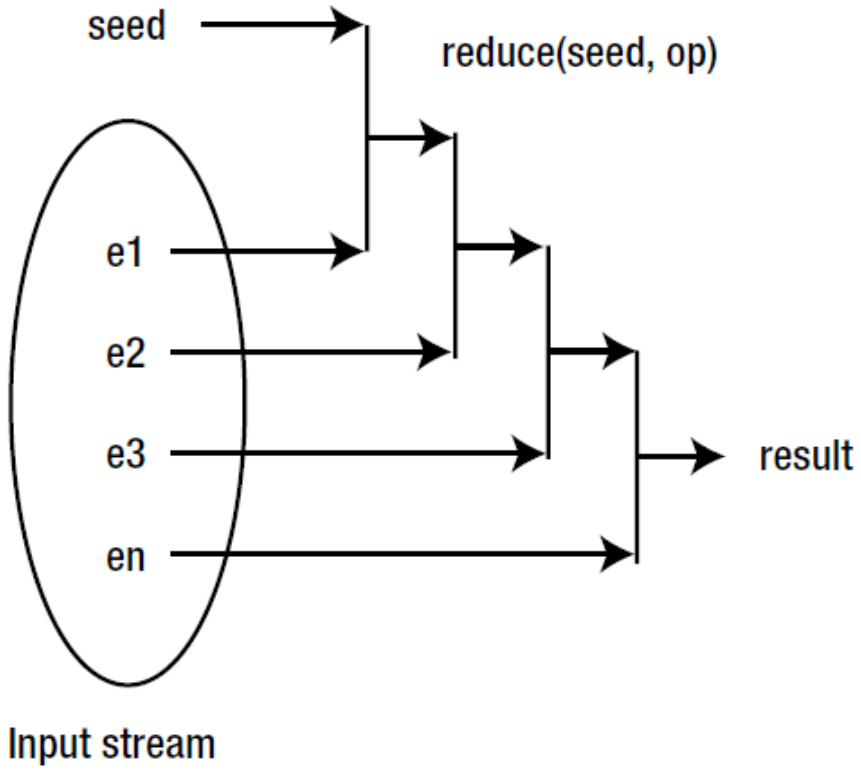
```
Stream<String> uniqueWords=
        Stream.of("merrily", "merrily", "merrily", "gently")
        .distinct();
// 得到的流中只会保留一个"merrily"
```

查找和匹配

- 使用allMatch()和anyMathch()方法
- 参看FindAndMatch.java

归约

- 使用reduce方法实现归约



理解归约-1

- 比如我们要统计一个集合中的元素个数

```
private static void countCollection(){
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    int sum = 0;
    for(int num : numbers) {
    sum = sum + num;
    }
    System.out.println(sum);
}
```

理解归约-2

- 使用归约的方法

```
private static void countCollection2() {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    int sum = numbers.stream().reduce(0, Integer::sum);
    System.out.println(sum);

}
```

上述reduce方法的第一个参数是初始值，称为seed，第二个参数是一个实现了BinaryOperator<T>的Lambda表达式，它接收的第一个数据元素就是seed，之后，就顺次接收流中的每个元素，每次都生成一个中间结果，将它与流中的下一个元素相加，直到遇到流中的最后一个元素，返回最终结果

统计功能

- 使用DoubleSummaryStatistics可以完成一些诸如最大值，最小值、平均值、和、个数的统计任务。
- 参看SummaryStats.java和IncomeStatsByGender.java

小结

- 本讲介绍了Java 8 Stream API的基础用法。
- 同学们重点要理解本讲中所介绍的Stream API的基本特性，在此基础上，再去逐步了解诸如filter(),map()之类的使用方法。掌握这些知识的前提是你已经掌握了Lambda表达式的基本编程技能，认识常用的函数式接口。
- 注意： Stream API是Java 8新特性中最重要的一个，它是所有Java程序员所必须掌握的基本技能。
- 本讲只是一个起点，更深入的内容就靠同学们自己探索了。