Lecture7 函数式编程与流编程

1. 流编程介绍

流与流处理

- 流 Stream：I/O 流，字符 / 字节流
- 流处理 Stream Processing：一种函数式的编程风格
  - 生产流：ofXXX, collection to stream, primitive streams
  - 转换流：map, filter, limit, skil, peek
  - 收集流：toXXX, collect, reduce, forEach

```
List<String> wordList = ...;
long count = 0;
for(String w : wordList){
    if(w.length() > 10) count++;
}
```

```
Stream<String> words = ...;
long count = words
    .filter(w -> w.length() > 10)
    .count();
```

`Stream` 听起来类似于 Java I/O 中的 `InputStream` 和 `OutputStream` 但它是完全不同的东西

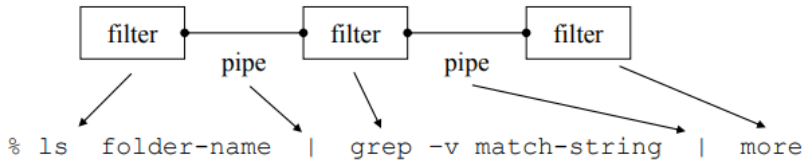
在函数式编程中，一个 `Monad` 是一个结构，它代表了定义为步骤序列的计算，一个类型与一个 `Monad` 结构定义它的意思是连锁操作，或嵌套的功能的类型在一起

Java 8 中的 `Stream` 写道

- Declarative — More concise and readable
- Composable — Greater flexibility
- Parallelizable — Better performance

管道和过滤器

- 组件：过滤器 filters 将输入转换为输出
- 连接器：通过管道 pipe 传输数据流
- 示例：UNIX 命令



流是如何工作的

中间操作和终端操作

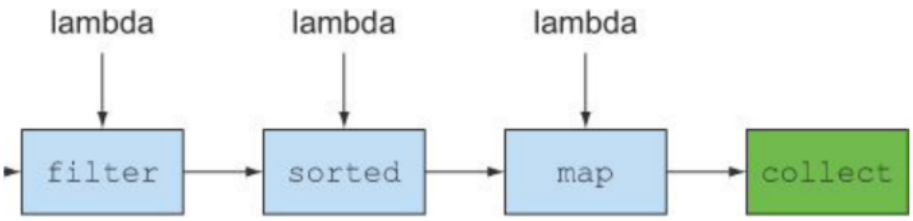
```
ArrayList<Element> arrayList = new ArrayList<Element>(n);
for (int i = 0; i < n; i++) {
 arrayList.add(new Element(i, in.nextInt()));
}
//input:
// value: 5 3 66 22 9 1 77 88 99
// index: 0 1 2 3 4 5 6 7 8
arrayList.stream().filter(e->e.value<50) //value: 5 3 22 9 1 index:0 1 3 4 5
    .map(e->new Element(e.index,e.value - e.index)) //value: 5 2 19 5 -4 index:0 1 3 4 5
    .sorted(methodList.get(method)) //value: -4 2 5 5 19 index:5 1 0 4 3
    .forEach(e->System.out.printf("%d ", e.index)); //print
```

流表示一个元素序列，并支持对这些元素执行不同类型的运算

流的操作分为**中间操作 intermediate** 和**终端操作 terminal**

- **中间操作**：返回一个流，所以我们可以链式连接一系列中间操作而不使用分号
  - filter, map, sorted
- **终端操作**：要么返回 `void`，要么返回一个不是流的结果
  - forEach

上面这些操作也被视为**操作管道 operation pipline**



函数式操作

大部分的流操作支持 lambda 表达式的参数，一种函数式接口来指定具体操作的行为，大部分操作必须要满足

- **不干涉的 non-interfering**
  - 流数据源在流处理中不可以被修改
  - 当一个函数不修改流的底层数据源时，它是不干涉的，例如在上面的例子中，没有 lambda 表达式通过添加或删除集合中的元素来修改 arrayList
- **无状态的 stateless**
  - 当操作的执行是确定的时，函数是无状态的，例如在上面的例子中，没有 lambda 表达式依赖于任何来自外部范围的可变变量或状态，这些变量或状态在执行过程中可能会改变

如何处理流操作

中间操作的一个重要特征是**懒惰性 laziness**

如下图所示的一段代码

```
Stream.of("CS", "209", "A").filter(s -> {
    System.out.println("filter: " + s);
    return true;
});
```

当执行这段代码时，控制台不会输出任何东西，这是因为**中间操作只会在终端操作出现时执行**

流的重用

流不能被重用，**只要你调用任何终端操作，流就会关闭**

```
Stream<String> stream = Stream.of("CS", "209", "A")
    .filter(s -> s.startsWith("C"));
stream.anyMatch(s -> true); // ok
stream.noneMatch(s -> true); // exceptio
```

同一个流上的 `anyMatch()` 之后调用 `nonemmatch()` 会导致异常

为了克服这个限制，我们必须为我们想要执行的每个终端操作创建一个新的流链，例如，我们可以创建一个 `Stream Supplier` 来构建一个新的流，所有中间操作都已经设置好了

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of("CS", "209", "A")
                                            .filter(s -> s.startsWith("C"));
streamSupplier.get().anyMatch(s -> true); // ok
streamSupplier.get().noneMatch(s -> true); // ok
```

2. 产生流 Producing Streams

Table 1 Producing Streams

Example	Result
`Stream.of(1, 2, 3)`	A stream containing the given elements. You can also pass an array.
`Collection<String> coll = . . .; coll.stream()`	A stream containing the elements of a collection.
`Files.lines(path)`	A stream of the lines in the file with the given path. Use a try-with-resources statement to ensure that the underlying file is closed.
`Stream<String> stream = . . .; stream.parallel()`	Turns a stream into a parallel stream.
`Stream.generate(() -> 1)`	An infinite stream of ones (see Special Topic 19.1).
`Stream.iterate(0, n -> n + 1)`	An infinite stream of `Integer` values (see Special Topic 19.1).
`IntStream.range(0, 100)`	An `IntStream` of `int` values between 0 (inclusive) and 100 (exclusive)—see Section 19.8.
`Random generator = new Random(); generator.ints(0, 100)`	An infinite stream of random `int` values drawn from a random generator—see Section 19.8.
`"Hello".codePoints()`	An `IntStream` of code points of a string—see Section 19.8.

创建固定数量的流

```java
// 创建一个固定数量的整数流
Stream.of(1,2,3)
    .forEach(System.out::println);

// 使用Stream.of()从一堆对象引用创建一个流
Stream.of("CS", "209", "A")
                .findFirst()
                .ifPresent(System.out::println);

// IntStreams 可以使用IntStream.range() 来替换常规的 for 循环
IntStream.range(1, 4)
                .forEach(System.out::println);
```

Array，ArrayList 转换成流

```java
// Array 转成流 此时 array 中的元素在流里面
int[] array = new int[]{1,2,3};
Arrays.stream(array).map(n->n*n).average().ifPresent(System.out::println);

// 把 array 本身当作流 array 对象在流中
Stream.of(array).map(n->{
            double sum = 0;
            for (int a:n) {
                sum += a*a;
            }
            return sum / n.length;
```

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

// 把 ArrayList 转成流
ArrayList<Integer> arrayList = new ArrayList<>();
arrayList.add(1);
arrayList.add(2);
arrayList.add(3);
arrayList.stream()
    .map(n -> n*n)
    .reduce((i,j) -> i+j)
    .ifPresent(System.out::println);
```

基本数据类型流与对象流的转换

```
// 对象流转换成 Int 流, Long 流,  Double 流等
// 使用 mapToInt(), mapToLong() and mapToDouble()
Stream.of("C1", "C2", "C3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);

// Int 流等也可以转换成对象流
IntStream.range(1, 4)
          .mapToObj(i -> "C" + i)
          .forEach(System.out::println);

// 也可以进行多次流转换
Stream.of(6.7, 8.7, 9.7)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "Level" + i)
    .forEach(System.out::println);
```

无限流 Infinite Streams

你可以通过 `generate()` 方法产生一个无限流

```
Stream<Integer> ones = Stream.generate(() -> 1); // 产生一个1的无限流
Stream<Integer> dieTosses = Stream.generate(() -> 1 + (int)(6 * Math.random())); // 产生一个随机产生数字1-6的无限流
Stream<Integer> integers = Stream.iterate(0, n -> n+1); // 产生一个从0开始逐渐累加的流
Stream<Integer> primes = integers.filter(n -> isPrime(n)); // 产生一个都是质数的流

// 生成随机整数的无限流
Random generator = new Random();
IntStream dieTosses = generator.ints(1,7);
```

当然，不能为这样的流生成所有元素

在某些情况下，需要限制结果，如果你想找到前 500 个质数，调用

```
List<Integer> firstPrimes = primes
    .limit(500)
    .collect(Collectors.toList());
```

使用无限流的优势是，即使它最终被截断为有限，你不需要提前知道要用多少个数才能得到所需的目标

3. 中间操作 - 流的转换

Table 3 Stream Transformations

Example	Comments
`stream.filter(condition)`	A stream with the elements matching the condition.
`stream.map(function)`	A stream with the results of applying the function to each element.

`stream.mapToInt(function)`
`stream.mapToDouble(function)`
`stream.mapToLong(function)`
	A primitive-type stream with the results of applying a function with a return value of a primitive type—see Section 19.8.

`stream.limit(n)` `stream.skip(n)`	A stream consisting of the first `n`, or all but the first `n` elements.

`stream.distinct()`
`stream.sorted()`
`stream.sorted(comparator)`
	A stream of the distinct or sorted elements from the original stream.

- filter(condition): 过滤只保留满足 condition 的部分

4. 终端操作- 收集流 Collecting Result

`Collect()` 是一个非常有用的终端操作，它将流元素转换成一个不同的结果，比如 `List`、`Set` 或者 `Map`

`Collect()` 方法接受一个 `Collector`，它支持四种不同的操作

- supplier
- accumulator
- conbiner
- finisher

不过 Java 8 在 `Collectors` 类中支持各种内置收集器，因此，对于最常见的操作，不必自己实现收集器

例如我们有一个 `Student` 类和 `College` 类

```
class Student {
```

```
    String name;
    int age;

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

```
class College {
    String collegeName;
    List<Student> students = new ArrayList<>();

    College(String name) {
        this.collegeName = name;
    }
}
```

将 `List<Student>` 构建流

```
List<Student> students =
    Arrays.asList(
        new Student("Zhang San", 17),
        new Student("Li si", 21),
        new Student("Wang Wu", 21),
        new Student("Liu Liu", 19));
```

从流构造出 List, Set

```
List<Student> filteredStudentsList = students
    .stream()
    .filter(s -> s.name.startsWith("L"))
    .collect(Collectors.toList()); // 如果需要一个 set, 使用 Collectors.toSet()

System.out.println(filteredStudentsList);
//output:
// [Li si, Liu Liu]
```

GroupBy

通过 `age` 将学生聚合

```
//groups all students by age
Map<Integer, List<Student>> studentsByAge = students
    .stream()
    .collect(Collectors.groupingBy(s -> s.age));

studentsByAge
    .forEach((age, s) -> System.out.format("age %s: %s\n", age, s));
//output:
//age 17: [Zhang San]
//age 19: [Liu Liu]
//age 21: [Li si, Wang Wu]
```

从流构造出基本数据类型

计算所有学生的平均年龄

```
Double averageAge = students
        .stream()
        .collect(Collectors.averagingInt(p -> p.age));
System.out.println(averageAge);
//output:
// 19.5
```

统计

Table 5 Computing Results from a Stream<T>

Example	Comments
`stream.count()`	Yields the number of elements as a `long` value.
`stream.findFirst()` `stream.findAny()`	Yields the first, or an arbitrary element as an `Optional<T>`—see Section 19.6.
`stream.max(`*comparator*`)` `stream.min(`*comparator*`)`	Yields the largest or smallest element as an `Optional<T>`—see Section 19.7.

`pstream.sum()`
`pstream.average()`
`pstream.max()`
`pstream.min()`
	The sum, average, maximum, or minimum of a primitive-type stream—see Section 19.8.

`stream.allMatch(`*condition*`)`
`stream.anyMatch(`*condition*`)`
`stream.noneMatch(`*condition*`)`
	Yields a `boolean` variable indicating whether all, any, or no elements match the condition—see Section 19.7.

`stream.forEach(`*action*`)`	Carries out the action on all stream elements—see Section 19.7.

SummaryStatistic

如果需要一个数值流的 count, sum, min, max ,average 等信息，可以考虑使用基本数据流中的 `summerStatistic()`

```
DoubleSummaryStatistics stats = DoubleStream.generate(Math::random)
    .limit(1_000_000)
    .summaryStatistics();

System.out.println(stats);  ❶

System.out.println("count: " + stats.getCount());
System.out.println("min  : " + stats.getMin());
System.out.println("max  : " + stats.getMax());
System.out.println("sum  : " + stats.getSum());
System.out.println("ave  : " + stats.getAverage());
```

如果是对象流，可以考虑使用 `Collectors.summarizingXXX()` 的方法

`Summarizing` 收集器返回一个特殊的内置汇总统计对象

你可以得到学生的数量，最大值，最小值，平均年龄等

```
IntSummaryStatistics ageSummary = students
    .stream()
    .collect(Collectors.summarizingInt(s -> s.age));

System.out.println(ageSummary)
//output:
// IntSummaryStatistics{count=4, sum=78, min=17, average=19.500000, max=21}
```

从流构造出字符串

连接收集器接受分隔符以及可选的前缀和后缀

该示例将所有成年的学生连接到一个字符串中

```
String phrase = students
        .stream()
        .filter(s -> s.age >= 18)
        .map(s -> s.name)
        .collect(Collectors.joining(" and ", "In China ", " are of legal age."));
System.out.println(phrase);
//output:
// In China Li si and Wang Wu and Liu Liu are of legal age.
```

从流构造出 Map

为了将一个流元素转换成一个 `Map`

需要同时指定映射的键和值

请记住，映射的键必须是唯一的，否则会抛出 `IllegalStateException`

```
Map<Integer, String> map = students
        .stream()
        .collect(Collectors.toMap(
                s -> s.age, // Function<? super T, ? extends K> keyMapper
                s -> s.name, // Function<? super T, ? extends U> valueMapper
                (name1, name2) -> name1 + "|" + name2)); // BinaryOperator<U>
mergeFunction
System.out.println(map);
//output:
// {17=Zhang San, 19=Liu Liu, 21=Li si|Wang Wu}
```

自定义 Collector

下面的示例将流的所有学生转换为由所有学生组成的单个字符串，由 `|` 管道字符分隔的大写字母名称

为了实现这个目标，我们要通过 `Collector.of()` 创建一个新的收集器

我们需要为一个收集器指定四个成分

- supplier
- accumulator
- conbiner
- finisher

由于在 Java 中 `String` 变量是不可变的，我们需要一个辅助类 `StringJoiner` 来让收集器产生一个字符串

- supplier 首先创建出一个具有适当分隔符的 `StringJoiner`
- accumulator 用来将每个人的大写名字添加到 `StringJoiner` 中
- combiner 知道如何将两个 `StringJoiner` 合并成一个
- finisher 构建一个最终的字符串

```
Collector<Student, StringJoiner, String> studentsNameCollector =
    Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, s) -> j.add(s.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher
```

```
String names = students
    .stream()
    .collect(studentsNameCollector);

System.out.println(names);
//output:
//ZHANG SAN | LI SI | WANG WU | LIU LIU
```

5. 终端操作 - Reduce

基本数据类型流的 reduce 操作

基本数据类型的流，如 `IntStream`、`LongStream` 和 `DoubleStream` 有很多在 API 内置的 reduce 操作

Method	Return type
average	OptionalDouble
count	long
max	OptionalInt
min	OptionalInt
sum	int
summaryStatistics	IntSummaryStatistics
collect(Supplier<R> supplier, ObjIntConsumer<R> accumulator, BiConsumer<R,R> combiner)	R
reduce	int, OptionalInt

```
String[] strings = "this is an array of strings".split(" ");
long count = Arrays.stream(strings)
        .map(String::length)          ❶
        .count();
System.out.println("There are " + count + " strings");

int totalLength = Arrays.stream(strings)
        .mapToInt(String::length)     ❷
        .sum();
System.out.println("The total length is " + totalLength);

OptionalDouble ave = Arrays.stream(strings)
        .mapToInt(String::length)     ❷
        .average();
System.out.println("The average length is " + ave);

OptionalInt max = Arrays.stream(strings)
        .mapToInt(String::length)     ❷
        .max();                       ❸

OptionalInt min = Arrays.stream(strings)
        .mapToInt(String::length)     ❷
        .min();                       ❸

System.out.println("The max and min lengths are " + max + " and " + min);
```

reduce 的三种方法

`reduce()` 操作将流的所有元素合并为一个结果，Java 8 支持三种不同的 Reduce 操作

第一种方法将一个流减少为只剩流中的一个元素

`reduce()` 方法接受 `BinaryOperator` 累加器函数式接口，它接受流中两个相同的元素，将它们以某种方式“合并”产生新的元素

下面的示例比较所有学生的年龄，返回年龄最大的学生

```
students
        .stream()
        .reduce((s1, s2) -> s1.age > s2.age ? s1 : s2)
        .ifPresent(System.out::println); //A student with the maxmum age
//Output:
//Wang Wu
```

第二种方法 `reduce()` 同时接受标识值和 `BinaryOperator` 累加器

该方法可用于构造一个同时集聚了姓名和年龄的新的 `Student` 对象

```
Student result =
        students
                .stream()
                // 第一个元素identity，  第二个元素 操作，累积的操作会放在identity里面
                .reduce(new Student("", 0), (s1, s2) -> {
                    // 对newStudent做累计
                    s1.age += s2.age;
                    s1.name = s1.name + "|" +s2.name;
                    return s1;
                });
```

```
 System.out.format("name=%s; age=%s\n", result.name, result.age);
//Output:
// name=|Zhang San|Li si|Wang Wu|Liu Liu; age=78
```

第三种 `reduce()` 方法接受三种参数值，一个标识值，一个 `BiFunction` 累加器和一个 `BinaryOperator` 结合器，它可以产生一个新的不同类型的对象

```
Integer ageSum = students
        .stream()
        .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);

System.out.println(ageSum);
//Output:
// 78
```

6. 其它的操作

flatMap

我们知道流中的 `map()` 方法将一种类型的流转换成另一种类型

假设存在一个流，这个流中的每一个对象里面都有一个 `List` 属性，我们想要把这个流里面的每个对象都的这个属性合并在一起处理一些事情

```
 List<College> collegeList = new ArrayList<>();

// create Colleges
IntStream
    .range(1, 3)
    .forEach(i -> collegeList.add(new College("College" + i)));

//add students
collegeList.forEach(f -> IntStream
     .range(0, 2)
     .forEach(i -> {
         int collegeNumber = Integer.parseInt(f.collegeName.substring(7));
         f.students.add(students.get((collegeNumber - 1) * 2 + i));
     }));
```

现在我们有一个 `List<College>` 对象，它有两个 `College` 对象

`flatMap()` 接受一个 `Function` 接口，它返回一个 `Stream` 的对象，来把这些流合并成一个流

```
collegeList.stream()
    .flatMap(f -> f.students.stream())
    .forEach(b -> System.out.println(b.name));

//output:
//Zhang San
//Li si
//Wang Wu
//Liu Liu
```

Optional 类

`Optional` 类是对可能存在也可能不存在的对象的包装

```
String result = oldFashionedMethod(searchParameters); // 如果不匹配返回为 null
int length = result.length(); // 如果 result 是 null，抛出 NullPointerException
```

如果进行流操作的时候

```
words.filter(w -> w.length() > 10).findFirst()
```

如果 words 中没有字符串长度大于 10 的元素呢?

考虑到这种情况，`findFirst()` 方法返回一个 `Optional<String>` 对象，而不是 `String`

```
Optional<String> optResult = words
    .filter(w -> w.length > 10)
    .findFirst();
```

使用 `orElse()` 方法提取值，或者使用提供的替代方法

```
int length = optResult.orElse("").length();
```

Table 4 Working with Optional Values

Example	Comments
`result = optional.orElse("");`	Extracts the wrapped value or the specified default if no value is present.
`optional.ifPresent(v -> Process v);`	Processes the wrapped value if present or does nothing if no value is present.

`if (optional.isPresent()) {`
`  Process optional.get()`
`} else {`
`  Handle the absence of a value.`
`}`
	Processes the wrapped value if present, or deals with the situation when it is not present.

`double average = pstream.average()` `  .getAsDouble();`	Gets the wrapped value from a primitive-type stream—see Section 19.8.

`if (there is a result) {`
`  return Optional.of(result);`
`} else {`
`  return Optional.empty();`
`}`
	Returns an `Optional` value from a method.

方法如 `findFirst()` 或者 `max()` 返回一个 `Optional` 结果，有可能没有结果

如果你不考虑存在空值的可能性而直接提取包装类里面的值，可能会出现异常 `NoSuchElementException`

最好的方法是避免调用 `Optional` 的 `get()` 方法，而用 `ifPresent()` 和 `orElse()` 进行替代

还是考虑之前的示例

```
String result = Optional<String> optResult = words
    .filter(w -> w.length > 10)
    .findFirst()
    .get(); // 如果值不存在，会抛出异常
```

如果想要避免这种情况，可以使用

```
String result = Optional<String> optResult = words
    .filter(w -> w.length > 10)
    .findFirst()
    .orElse("None");
System.out.println("Long word: " + result);
```

如果你想要只在 `Optional` 对象不是空的时候，对它进行某些操作，可以使用 `ifPresent()`

```
List<String> results = new ArrayList<String>;
words.stream()
    .filter(w -> w.length > 10)
    .findFirst()
    .ifPresent(w -> results.add(w))
```

当然，也可以针对 `Optional` 为空或者不为空做出不同的处理操作

```
Optional<String> result = stream
    .filter(w -> w.length() > 10)
    .findFirst();

if(result.isPresent()){
    results.add(result.get()); // 安全的调用 get()
}else{
    System.out.println("No long words")
}
```