链滴

Stream 流编程

作者：lonelyant
原文链接：https://ld246.com/article/1556413830022
来源网站：链滴

概念

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。

以上取自菜鸟教程

在接触Stream以前，我们想要对一个数组进行迭代，使用的是如下方法（外部迭代）

```
int[] nums = { 1, 2, 3 };
// 外部迭代
int sum = 0;
for (int i : nums) {
    sum += I;
}
```

而在1.8以后，我们可以使用**内部迭代**

```
// 使用stream的内部迭代
// sum就是终止操作
int sum2 = IntStream.of(nums).sum();
```

流的创建

Stream流编程 – 创建

	相关方法
集合	Collection.stream/parallelStream
数组	Arrays.stream
数字Stream	IntStream/LongStream. range/rangeClosed
	Random.ints/longs/doubles
自己创建	Stream. generate/iterate

```
List<String> list = new ArrayList<>();

// 从集合创建
list.stream();
list.parallelStream();

// 从数组创建
Arrays.stream(new int[] { 2, 3, 5 });

// 创建数字流
IntStream.of(1, 2, 3);
IntStream.rangeClosed(1, 10);

// 使用random创建一个无限流
new Random().ints().limit(10);
Random random = new Random();

// 自己产生流
Stream.generate(() -> random.nextInt()).limit(20);
```

注意：针对于无限流，需要使用limit限流，不然会报错

中间操作

Stream流编程 – 中间操作

	相关方法
无状态操作	map/ mapToXxx
	flatMap/ flatMapToXxx
	filter
	peek
	unordered
有状态操作	distinct
	sorted
	limit / skip

中间操作包含两类，无状态操作与有状态操作。

无状态操作是可在元素上执行而无需知道其他任何元素的操作。例如，过滤操作只需检查当前元素来定是包含还是消除它，但排序操作必须查看所有元素之后才知道首先发出哪个元素。

中间操作常用方法

filter、map

```
String str = "my name is 007";

// 打印字符长度大于2的单词长度
Stream.of(str.split(" ")).filter(s -> s.length() > 2)
        .map(s -> s.length()).forEach(System.out::println);
```

flatMap

```
// flatMap A->B属性(是个集合), 最终得到所有的A元素里面的所有B属性集合
// intStream/longStream 并不是Stream的子类, 所以要进行装箱 boxed
Stream.of(str.split(" ")).flatMap(s -> s.chars().boxed())
        .forEach(i -> System.out.println((char) i.intValue()));
```

peek

```
// peek 用于debug. 是个中间操作，forEach 是终止操作
Stream.of(str.split(" ")).peek(System.out::println)
        .forEach(System.out::println);
```

limit

```
// limit用来对无限流限流
new Random().ints().filter(i -> i > 100 && i < 1000).limit(10)
        .forEach(System.out::println);
```

终止操作

Stream流编程 – 终止操作

	相关方法
非短路操作	forEach/ forEachOrdered
	collect/ toArray
	reduce
	min/ max/ count
短路操作	findFirst/ findAny
	allMatch/ anyMatch/ noneMatch

终止操作包含非短路操作和短路操作。

短路操作为不需要等待所有结果都计算完即可返回结果，比如findFirst，拿到第一个元素就可以结束作。非短路操作则要等到所有元素都计算完成才会结束。

终止操作常用方法

```
String str = "my name is 007";
```

forEachOrdered

```
str.chars().parallel().forEach(i -> System.out.print((char) i));
System.out.println();
// 使用 forEachOrdered 保证顺序
str.chars().parallel().forEachOrdered(i -> System.out.print((char) i));
```

使用并行流的时候，单纯的使用forEach方法去打印，会乱序；使用forEachOrdered能保证顺序。

collect

```
// 收集到list
List<String> list = Stream.of(str.split(" ")).collect(Collectors.toList());
```

reduce

```
// 使用 reduce 拼接字符串
Optional<String> letters = Stream.of(str.split(" ")).reduce((s1, s2) -> s1 + "|" + s2);
System.out.println(letters.orElse(""));

// 带初始化值的reduce
String reduce = Stream.of(str.split(" ")).reduce("",(s1, s2) -> s1 + "|" + s2);
System.out.println(reduce);

// 计算所有单词总长度
Integer length = Stream.of(str.split(" ")).map(s -> s.length()).reduce(0, (s1, s2) -> s1 + s2);
System.out.println(length);
```

max

```
// max 的使用
Optional<String> max = Stream.of(str.split(" ")).max((s1, s2) -> s1.length() - s2.length());
System.out.println(max.get());
```

findFirst

```
// 使用 findFirst 短路操作
OptionalInt findFirst = new Random().ints().findFirst();
System.out.println(findFirst.getAsInt());
```

并行流

首先定义一个静态方法，方便测试

```
public class Demo {
    public static void debug(int i) {
        System.out.println(Thread.currentThread().getName() + " debug " + i);
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

基本并行流

```
// 调用parallel 产生一个并行流
IntStream.range(1, 10).parallel().peek(Demo::debug).count();
```

结尾调用count终止操作，如果不调用终止操作，那么中间操作不会被执行。

parallel / sequential

```
 IntStream.range(1, 100)
```

```
 // 调用parallel产生并行流
 .parallel().peek(Demo::debug)
 // 调用sequential 产生串行流
 .sequential().peek(Demo::debug2)
 .count();
```

多次调用 parallel / sequential, 整体执行方式是串行还是并行，以最后一次调用为准。

并行流使用的线程池

并行流使用的线程池: ForkJoinPool.commonPool

默认的线程数是 当前机器的cpu个数，允许修改默认线程数

```
// 修改默认线程数为20
 System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism",
 "20");
 IntStream.range(1, 100).parallel().peek(Demo::debug).count();
```

并行流使用自定义线程池

```
// 使用自己的线程池, 不使用默认线程池, 防止任务被阻塞
// 线程名字 : ForkJoinPool-1
ForkJoinPool pool = new ForkJoinPool(20);
pool.submit(() -> IntStream.range(1, 100).parallel()
        .peek(Demo::debug).count());
```

收集器

Stream流编程里面收集器是很重要的一部分，这里先只放一个简单例子，以后有机会再深入研究。

```
/**
 * 学生 对象
 */
class Student {
    /**
     * 姓名
     */
    private String name;

    /**
     * 年龄
     */
    private int age;

    /**
     * 性别
     */
    private Gender gender;

    /**
     * 班级
     */
```

```
    private Grade grade;

    public Student(String name, int age, Gender gender, Grade grade) {
        super();
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.grade = grade;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Grade getGrade() {
        return grade;
    }

    public void setGrade(Grade grade) {
        this.grade = grade;
    }

    public Gender getGender() {
        return gender;
    }

    public void setGender(Gender gender) {
        this.gender = gender;
    }

    @Override
    public String toString() {
        return "[name=" + name + ", age=" + age + ", gender=" + gender
            + ", grade=" + grade + "]";
    }
}

/**
 * 性别
 */
enum Gender {
    MALE, FEMALE
```

```java
}

/**
 * 班级
 */
enum Grade {
    ONE, TWO, THREE, FOUR;
}

public class CollectDemo {

    public static void main(String[] args) {
        // 测试数据
        List<Student> students = Arrays.asList(
            new Student("小明", 10, Gender.MALE, Grade.ONE),
            new Student("大明", 9, Gender.MALE, Grade.THREE),
            new Student("小白", 8, Gender.FEMALE, Grade.TWO),
            new Student("小黑", 13, Gender.FEMALE, Grade.FOUR),
            new Student("小红", 7, Gender.FEMALE, Grade.THREE),
            new Student("小黄", 13, Gender.MALE, Grade.ONE),
            new Student("小青", 13, Gender.FEMALE, Grade.THREE),
            new Student("小紫", 9, Gender.FEMALE, Grade.TWO),
            new Student("小王", 6, Gender.MALE, Grade.ONE),
            new Student("小李", 6, Gender.MALE, Grade.ONE),
            new Student("小马", 14, Gender.FEMALE, Grade.FOUR),
            new Student("小刘", 13, Gender.MALE, Grade.FOUR));

        // 得到所有学生的年龄列表
        // s -> s.getAge() --> Student::getAge , 不会多生成一个类似 lambda$0这样的函数
        Set<Integer> ages = students.stream().map(Student::getAge)
            .collect(Collectors.toCollection(TreeSet::new));
        System.out.println("所有学生的年龄:" + ages);

        // 统计汇总信息
        IntSummaryStatistics agesSummaryStatistics = students.stream()
            .collect(Collectors.summarizingInt(Student::getAge));
        System.out.println("年龄汇总信息:" + agesSummaryStatistics);

        // 分块
        Map<Boolean, List<Student>> genders = students.stream().collect(
            Collectors.partitioningBy(s -> s.getGender() == Gender.MALE));
        System.out.println("男女学生列表:" + genders);

        // 分组
        Map<Grade, List<Student>> grades = students.stream()
            .collect(Collectors.groupingBy(Student::getGrade));
        System.out.println("学生班级列表:" + genders);

        // 得到所有班级学生的个数
        Map<Grade, Long> gradesCount = students.stream().collect(Collectors
            .groupingBy(Student::getGrade, Collectors.counting()));
        System.out.println("班级学生个数列表:" + genders);
    }
```

```
}
```