Stream API

Java从8开始，不但引入了Lambda表达式，还引入了一个全新的流式API：Stream API。它位于java.util.stream包中。

**Stream API的特点是：**

* Stream API提供了一套新的流式处理的抽象序列；
* Stream API支持函数式编程和链式操作；
* Stream可以表示无限序列，并且大多数情况下是惰性求值的。

*划重点*：这个Stream不同于java.io的InputStream和OutputStream，它代表的是任意Java对象的序列。两者对比如下：

|  | **java.io** | **java.util.stream** |
| --- | --- | --- |
| 存储 | 顺序读写的byte或char | 顺序输出的任意Java对象实例 |
| 用途 | 序列化至文件或网络 | 内存计算／业务逻辑 |

有同学会问：一个顺序输出的Java对象序列，不就是一个List容器吗？

*再次划重点*：**这个Stream和List也不一样，List存储的每个元素都是已经存储在内存中的某个Java对象，而Stream输出的元素可能并没有预先存储在内存中，而是实时计算出来的**。

换句话说，List的用途是操作一组已存在的Java对象，而Stream实现的是惰性计算，两者对比如下：

|  | **java.util.List** | **java.util.stream** |
| --- | --- | --- |
| 元素 | 已分配并存储在内存 | 可能未分配，实时计算 |
| 用途 | 操作一组已存在的Java对象 | 惰性计算 |

Stream看上去有点不好理解，但我们举个例子就明白了。

如果我们要表示一个全体自然数的集合，显然，用List是不可能写出来的，因为自然数是无限的，内存再大也没法放到List中：

List<BigInteger> list = ??? // 全体自然数?

但是，用Stream可以做到。写法如下：

Stream<BigInteger> naturals = createNaturalStream(); // 全体自然数

我们先不考虑createNaturalStream()这个方法是如何实现的，我们看看如何使用这个Stream。

首先，我们可以对每个自然数做一个平方，这样我们就把这个Stream转换成了另一个Stream：

Stream<BigInteger> naturals = createNaturalStream(); // 全体自然数

Stream<BigInteger> streamNxN = naturals.map(n -> n.multiply(n)); // 全体自然数的平方

因为这个streamNxN也有无限多个元素，要打印它，必须首先把无限多个元素变成有限个元素，可以用limit()方法截取前100个元素，最后用forEach()处理每个元素，这样，我们就打印出了前100个自然数的平方：

Stream<BigInteger> naturals = createNaturalStream();

naturals.map(n -> n.multiply(n)) // 1, 4, 9, 16, 25...

.limit(100)

.forEach(System.out::println);

**我们总结一下Stream的特点：它可以“存储”有限个或无限个元素。这里的存储打了个引号，是因为元素有可能已经全部存储在内存中，也有可能是根据需要实时计算出来的。**

**Stream的另一个特点是，一个Stream可以轻易地转换为另一个Stream，而不是修改原Stream本身。**

最后，真正的计算通常发生在最后结果的获取，也就是**惰性计算**。

Stream<BigInteger> naturals = createNaturalStream(); // 不计算

Stream<BigInteger> s2 = naturals.map(BigInteger::multiply); // 不计算

Stream<BigInteger> s3 = s2.limit(100); // 不计算

s3.forEach(System.out::println); // 计算

**惰性计算的特点是：一个Stream转换为另一个Stream时，实际上只存储了转换规则，并没有任何计算发生。**

例如，创建一个全体自然数的Stream，不会进行计算，把它转换为上述s2这个Stream，也不会进行计算。再把s2这个无限Stream转换为s3这个有限的Stream，也不会进行计算。只有最后，调用forEach确实需要Stream输出的元素时，才进行计算。我们通常把Stream的操作写成链式操作，代码更简洁：

createNaturalStream()

.map(BigInteger::multiply)

.limit(100)

.forEach(System.out::println);

因此，Stream API的基本用法就是：创建一个Stream，然后做若干次转换，最后调用一个求值方法获取真正计算的结果：

int result = createNaturalStream() // 创建Stream

.filter(n -> n % 2 == 0) // 任意个转换

.map(n -> n \* n) // 任意个转换

.limit(100) // 任意个转换

.sum(); // 最终计算结果

## 创建Stream

要使用Stream，就必须先创建它。创建Stream有很多种方法，我们来一一介绍。

**Stream.of()**

创建Stream最简单的方式是直接用Stream.of()静态方法，传入可变参数即创建了一个能输出确定元素的Stream：

public class Main {

public static void main(String[] args) {

**Stream<String> stream = Stream.of("A", "B", "C", "D");**

// forEach()方法相当于内部循环调用，

// 可传入符合Consumer接口的void accept(T t)的方法引用：

stream.forEach(System.out::println);

}

}

虽然这种方式基本上没啥实质性用途，但**测试**的时候很方便。

**基于数组或Collection**

第二种创建Stream的方法是基于一个数组或者Collection，这样该Stream输出的元素就是数组或者Collection持有的元素：

public class Main {

public static void main(String[] args) {

Stream<String> stream1 = Arrays.stream(new String[] { "A", "B", "C" });

Stream<String> stream2 = List.of("X", "Y", "Z").stream();

stream1.forEach(System.out::println);

stream2.forEach(System.out::println);

}

}

把数组变成Stream使用Arrays.stream()方法。对于Collection（List、Set、Queue等），直接调用stream()方法就可以获得Stream。

上述创建Stream的方法都是把一个现有的序列变为Stream，它的元素是固定的。

**基于Supplier**

创建Stream还可以通过Stream.generate()方法，它需要传入一个Supplier对象：

Stream<String> s = Stream.generate(Supplier<String> sp);

基于Supplier创建的Stream会不断调用Supplier.get()方法来不断产生下一个元素，这种Stream保存的不是元素，而是算法，它可以用来表示无限序列。

例如，我们编写一个能不断生成自然数的Supplier，它的代码非常简单，每次调用get()方法，就生成下一个自然数：

public class Main {

public static void main(String[] args) {

Stream<Integer> natual = Stream.generate(new NatualSupplier());

// 注意：无限序列必须先变成有限序列再打印:

natual.limit(20).forEach(System.out::println);

}

}

class NatualSupplier implements Supplier<Integer> {

int n = 0;

public Integer get() {

n++;

return n;

}

}

上述代码我们用一个Supplier<Integer>模拟了一个无限序列（当然受int范围限制不是真的无限大）。如果用List表示，即便在int范围内，也会占用巨大的内存，而Stream几乎不占用空间，因为每个元素都是实时计算出来的，用的时候再算。

对于无限序列，如果直接调用forEach()或者count()这些最终求值操作，会进入死循环，因为永远无法计算完这个序列，所以正确的方法是先把无限序列变成有限序列，例如，用limit()方法可以截取前面若干个元素，这样就变成了一个有限序列，对这个有限序列调用forEach()或者count()操作就没有问题。

**其他方法**

创建Stream的第三种方法是通过一些API提供的接口，直接获得Stream。

例如，Files类的lines()方法可以把一个文件变成一个Stream，每个元素代表文件的一行内容：

try (Stream<String> lines = Files.lines(Paths.get("/path/to/file.txt"))) {

...

}

此方法对于按行遍历文本文件十分有用。

另外，正则表达式的Pattern对象有一个splitAsStream()方法，可以直接把一个长字符串分割成Stream序列而不是数组：

Pattern p = Pattern.compile("\\s+");

Stream<String> s = p.splitAsStream("The quick brown fox jumps over the lazy dog");

s.forEach(System.out::println);

**基本类型**

因为Java的范型不支持基本类型，所以我们无法用Stream<int>这样的类型，会发生编译错误。为了保存int，只能使用Stream<Integer>，但这样会产生频繁的装箱、拆箱操作。为了提高效率，Java标准库提供了IntStream、LongStream和DoubleStream这三种使用基本类型的Stream，它们的使用方法和范型Stream没有大的区别，设计这三个Stream的目的是提高运行效率：

// 将int[]数组变为IntStream:

IntStream is = Arrays.stream(new int[] { 1, 2, 3 });

// 将Stream<String>转换为LongStream:

LongStream ls = List.of("1", "2", "3").stream().mapToLong(Long::parseLong);

**练习**

编写一个能输出斐波拉契数列（Fibonacci）的LongStream：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

**小结**

创建Stream的方法有 ：

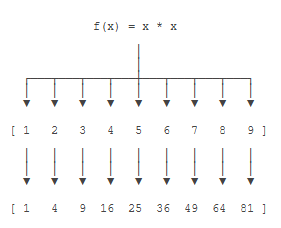
* **通过指定元素、指定数组、指定Collection创建Stream；**
* 通过Supplier创建Stream，可以是无限序列；
* 通过其他类的相关方法创建。

基本类型的Stream有**IntStream**、**LongStream**和**DoubleStream**。

## 使用map

Stream.map()是Stream最常用的一个转换方法，它把一个Stream转换为另一个Stream。

所谓map操作，就是把一种操作运算，映射到一个序列的每一个元素上。例如，对x计算它的平方，可以使用函数f(x) = x \* x。我们把这个函数映射到一个序列1，2，3，4，5上，就得到了另一个序列1，4，9，16，25：



可见，map操作，把一个Stream的每个元素一一对应到应用了目标函数的结果上。

Stream<Integer> s = Stream.of(1, 2, 3, 4, 5);

Stream<Integer> s2 = s.map(n -> n \* n);

如果我们查看Stream的源码，会发现map()方法接收的对象是Function接口对象，它定义了一个apply()方法，负责把一个T类型转换成R类型：

<R> Stream<R> map(Function<? super T, ? extends R> mapper);

其中，Function的定义是：

@FunctionalInterface

public interface Function<T, R> {

// 将T类型转换为R:

R apply(T t);

}

利用map()，不但能完成数学计算，对于字符串操作，以及任何Java对象都是非常有用的。例如：

public class Main {

public static void main(String[] args) {

List.of(" Apple ", " pear ", " ORANGE", " BaNaNa ")

.stream()

.map(String::trim) // 去空格

.map(String::toLowerCase) // 变小写

.forEach(System.out::println); // 打印

}

}

通过若干步map转换，可以写出逻辑简单、清晰的代码。

**练习**

使用map()把一组String转换为LocalDate并打印。

**小结**

map()方法用于将一个Stream的每个元素映射成另一个元素并转换成一个新的Stream；

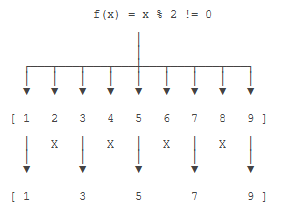
可以将一种元素类型转换成另一种元素类型。

## 使用filter

Stream.filter()是Stream的另一个常用转换方法。

所谓filter()操作，就是对一个Stream的所有元素一一进行测试，不满足条件的就被“滤掉”了，剩下的满足条件的元素就构成了一个新的Stream。

例如，我们对1，2，3，4，5这个Stream调用filter()，传入的测试函数f(x) = x % 2 != 0用来判断元素是否是奇数，这样就过滤掉偶数，只剩下奇数，因此我们得到了另一个序列1，3，5：



用IntStream写出上述逻辑，代码如下：

public class Main {

public static void main(String[] args) {

IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)

.filter(n -> n % 2 != 0)

.forEach(System.out::println);

}

}

从结果可知，经过filter()后生成的Stream元素可能变少。

filter()方法接收的对象是Predicate接口对象，它定义了一个test()方法，负责判断元素是否符合条件：

@FunctionalInterface

public interface Predicate<T> {

// 判断元素t是否符合条件:

boolean test(T t);

}

filter()除了常用于数值外，也可应用于任何Java对象。例如，从一组给定的LocalDate中过滤掉工作日，以便得到休息日：

public class Main {

public static void main(String[] args) {

Stream.generate(new LocalDateSupplier())

.limit(31)

.filter(ldt -> ldt.getDayOfWeek() == DayOfWeek.SATURDAY || ldt.getDayOfWeek() == DayOfWeek.SUNDAY)

.forEach(System.out::println);

}

}

class LocalDateSupplier implements Supplier<LocalDate> {

LocalDate start = LocalDate.of(2020, 1, 1);

int n = -1;

public LocalDate get() {

n++;

return start.plusDays(n);

}

}

**练习**

请使用filter()过滤出成绩及格的同学，并打印出名字。

**小结**

使用filter()方法可以对一个Stream的每个元素进行测试，通过测试的元素被过滤后生成一个新的Stream。

## 使用reduce

map()和filter()都是Stream的转换方法，而Stream.reduce()则是Stream的一个聚合方法，它可以把一个Stream的所有元素按照聚合函数聚合成一个结果。

我们来看一个简单的聚合方法：

public class Main {

public static void main(String[] args) {

int sum = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).reduce(0, (acc, n) -> acc + n);

System.out.println(sum); // 45

}

}

reduce()方法传入的对象是BinaryOperator接口，它定义了一个apply()方法，负责把上次累加的结果和本次的元素 进行运算，并返回累加的结果：

@FunctionalInterface

public interface BinaryOperator<T> {

// Bi操作：两个输入，一个输出

T apply(T t, T u);

}

上述代码看上去不好理解，但我们用for循环改写一下，就容易理解了：

Stream<Integer> stream = ...

int sum = 0;

for (n : stream) {

sum = (sum, n) -> sum + n;

}

可见，reduce()操作首先初始化结果为指定值（这里是0），紧接着，reduce()对每个元素依次调用(acc, n) -> acc + n，其中，acc是上次计算的结果：

// 计算过程:

acc = 0 // 初始化为指定值

acc = acc + n = 0 + 1 = 1 // n = 1

acc = acc + n = 1 + 2 = 3 // n = 2

acc = acc + n = 3 + 3 = 6 // n = 3

acc = acc + n = 6 + 4 = 10 // n = 4

acc = acc + n = 10 + 5 = 15 // n = 5

acc = acc + n = 15 + 6 = 21 // n = 6

acc = acc + n = 21 + 7 = 28 // n = 7

acc = acc + n = 28 + 8 = 36 // n = 8

acc = acc + n = 36 + 9 = 45 // n = 9

因此，实际上这个reduce()操作是一个求和。

如果去掉初始值，我们会得到一个Optional<Integer>：

Optional<Integer> opt = stream.reduce((acc, n) -> acc + n);

if (opt.isPresent()) {

System.out.println(opt.get());

}

这是因为Stream的元素有可能是0个，这样就没法调用reduce()的聚合函数了，因此返回Optional对象，需要进一步判断结果是否存在。

利用reduce()，我们可以把求和改成求积，代码也十分简单：

public class Main {

public static void main(String[] args) {

int s = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).reduce(1, (acc, n) -> acc \* n);

System.out.println(s); // 362880

}

}

注意：计算求积时，初始值必须设置为1。

除了可以对数值进行累积计算外，灵活运用reduce()也可以对Java对象进行操作。下面的代码演示了如何将配置文件的每一行配置通过map()和reduce()操作聚合成一个Map<String, String>：

public class Main {

public static void main(String[] args) {

// 按行读取配置文件:

List<String> props = List.of("profile=native", "debug=true", "logging=warn", "interval=500");

Map<String, String> map = props.stream()

// 把k=v转换为Map[k]=v:

.map(kv -> {

String[] ss = kv.split("\\=", 2);

return Map.of(ss[0], ss[1]);

})

// 把所有Map聚合到一个Map:

.reduce(new HashMap<String, String>(), (m, kv) -> {

m.putAll(kv);

return m;

});

// 打印结果:

map.forEach((k, v) -> {

System.out.println(k + " = " + v);

});

}

}

**小结**

reduce()方法将一个Stream的每个元素依次作用于BinaryOperator，并将结果合并。

reduce()是聚合方法，聚合方法会立刻对Stream进行计算。

## 输出集合

我们介绍了Stream的几个常见操作：map()、filter()、reduce()。这些操作对Stream来说可以分为两类，一类是转换操作，即把一个Stream转换为另一个Stream，例如map()和filter()，另一类是聚合操作，即对Stream的每个元素进行计算，得到一个确定的结果，例如reduce()。

### 输出为List

reduce()只是一种聚合操作，如果我们希望把Stream的元素保存到集合，例如List，因为List的元素是确定的Java对象，因此，把Stream变为List不是一个转换操作，而是一个聚合操作，它会强制Stream输出每个元素。

下面的代码演示了如何将一组String先过滤掉空字符串，然后把非空字符串保存到List中：

public class Main {

public static void main(String[] args) {

Stream<String> stream = Stream.of("Apple", "", null, "Pear", " ", "Orange");

List<String> list = stream.filter(s -> s != null && !s.isBlank()).collect(Collectors.toList());

System.out.println(list);

}

}

把Stream的每个元素收集到List的方法是调用collect()并传入Collectors.toList()对象，它实际上是一个Collector实例，通过类似reduce()的操作，把每个元素添加到一个收集器中（实际上是ArrayList）。

类似的，collect(Collectors.toSet())可以把Stream的每个元素收集到Set中。

### 输出为数组

把Stream的元素输出为数组和输出为List类似，我们只需要调用toArray()方法，并传入数组的“构造方法”：

List<String> list = List.of("Apple", "Banana", "Orange");

String[] array = list.stream().toArray(String[]::new);

注意到传入的“构造方法”是String[]::new，它的签名实际上是IntFunction<String[]>定义的String[] apply(int)，即传入int参数，获得String[]数组的返回值。

### 输出为Map

如果我们要把Stream的元素收集到Map中，就稍微麻烦一点。因为对于每个元素，添加到Map时需要key和value，因此，我们要指定两个映射函数，分别把元素映射为key和value：

public class Main {

public static void main(String[] args) {

Stream<String> stream = Stream.of("APPL:Apple", "MSFT:Microsoft");

Map<String, String> map = stream

.collect(Collectors.toMap(

// 把元素s映射为key:

s -> s.substring(0, s.indexOf(':')),

// 把元素s映射为value:

s -> s.substring(s.indexOf(':') + 1)));

System.out.println(map);

}

}

## 分组输出

Stream还有一个强大的分组功能，可以按组输出。我们看下面的例子：

public class Main {

public static void main(String[] args) {

List<String> list = List.of("Apple", "Banana", "Blackberry", "Coconut", "Avocado", "Cherry", "Apricots");

Map<String, List<String>> groups = list.stream()

.collect(Collectors.groupingBy(s -> s.substring(0, 1), Collectors.toList()));

System.out.println(groups);

}

}

分组输出使用Collectors.groupingBy()，它需要提供两个函数：一个是分组的key，这里使用s -> s.substring(0, 1)，表示只要首字母相同的String分到一组，第二个是分组的value，这里直接使用Collectors.toList()，表示输出为List，上述代码运行结果如下：

{

A=[Apple, Avocado, Apricots],

B=[Banana, Blackberry],

C=[Coconut, Cherry]

}

可见，结果一共有3组，按"A"，"B"，"C"分组，每一组都是一个List。

假设有这样一个Student类，包含学生姓名、班级和成绩：

class Student {

int gradeId; // 年级

int classId; // 班级

String name; // 名字

int score; // 分数

}

如果我们有一个Stream<Student>，利用分组输出，可以非常简单地按年级或班级把Student归类。

**小结**

Stream可以输出为集合：

Stream通过collect()方法可以方便地输出为List、Set、Map，还可以分组输出。

## 其他操作

我们把Stream提供的操作分为两类：转换操作和聚合操作。除了前面介绍的常用操作外，Stream还提供了一系列非常有用的方法。

### 排序

对Stream的元素进行排序十分简单，只需调用sorted()方法：

public class Main {

public static void main(String[] args) {

List<String> list = List.of("Orange", "apple", "Banana")

.stream()

**.sorted()**

.collect(Collectors.toList());

System.out.println(list);

}

}

此方法要求Stream的每个元素必须实现Comparable接口。如果要自定义排序，传入指定的Comparator即可：

List<String> list = List.of("Orange", "apple", "Banana")

.stream()

**.sorted(String::compareToIgnoreCase)**

.collect(Collectors.toList());

注意sorted()只是一个转换操作，它会返回一个新的Stream。

### 去重

对一个Stream的元素进行去重，没必要先转换为Set，可以直接用distinct()：

List.of("A", "B", "A", "C", "B", "D")

.stream()

**.distinct()**

.collect(Collectors.toList()); // [A, B, C, D]

### 截取

截取操作常用于把一个无限的Stream转换成有限的Stream，skip()用于跳过当前Stream的前N个元素，limit()用于截取当前Stream最多前N个元素：

List.of("A", "B", "C", "D", "E", "F")

.stream()

**.skip(2) // 跳过A, B**

**.limit(3) // 截取C, D, E**

.collect(Collectors.toList()); // [C, D, E]

截取操作也是一个转换操作，将返回新的Stream。

### 合并

将两个Stream合并为一个Stream可以使用Stream的静态方法concat()：

Stream<String> s1 = List.of("A", "B", "C").stream();

Stream<String> s2 = List.of("D", "E").stream();

// 合并:

Stream<String> s = Stream.**concat**(s1, s2);

System.out.println(s.collect(Collectors.toList())); // [A, B, C, D, E]

### flatMap

如果Stream的元素是集合：

Stream<List<Integer>> s = Stream.of(

Arrays.asList(1, 2, 3),

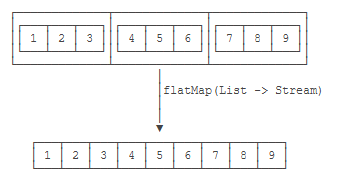
Arrays.asList(4, 5, 6),

Arrays.asList(7, 8, 9));

而我们希望把上述Stream转换为Stream<Integer>，就可以使用flatMap()：

Stream<Integer> i = s.flatMap(list -> list.stream());

因此，所谓flatMap()，是指把Stream的每个元素（这里是List）映射为Stream，然后合并成一个新的Stream：



### 并行

通常情况下，对Stream的元素进行处理是单线程的，即一个一个元素进行处理。但是很多时候，我们希望可以并行处理Stream的元素，因为在元素数量非常大的情况，并行处理可以大大加快处理速度。

把一个普通Stream转换为可以并行处理的Stream非常简单，只需要用parallel()进行转换：

Stream<String> s = ...

String[] result = s.parallel() // 变成一个可以并行处理的Stream

.sorted() // 可以进行并行排序

.toArray(String[]::new);

经过parallel()转换后的Stream只要可能，就会对后续操作进行并行处理。我们不需要编写任何多线程代码就可以享受到并行处理带来的执行效率的提升。

### 其他聚合方法

除了reduce()和collect()外，Stream还有一些常用的聚合方法：

* count()：用于返回元素个数；
* max(Comparator<? super T> cp)：找出最大元素；
* min(Comparator<? super T> cp)：找出最小元素。

针对IntStream、LongStream和DoubleStream，还额外提供了以下聚合方法：

* sum()：对所有元素求和；
* average()：对所有元素求平均数。

还有一些方法，用来测试Stream的元素是否满足以下条件：

* boolean allMatch(Predicate<? super T>)：测试是否所有元素均满足测试条件；
* boolean anyMatch(Predicate<? super T>)：测试是否至少有一个元素满足测试条件。

最后一个常用的方法是forEach()，它可以循环处理Stream的每个元素，我们经常传入System.out::println来打印Stream的元素：

Stream<String> s = ...

s.forEach(str -> {

System.out.println("Hello, " + str);

});

**小结**

Stream提供的常用操作有：

转换操作：map()，filter()，sorted()，distinct()；

合并操作：concat()，flatMap()；

并行处理：parallel()；

聚合操作：reduce()，collect()，count()，max()，min()，sum()，average()；

其他操作：allMatch(), anyMatch(), forEach()。