1、什么是流?

Stream 作为 Java 8 的一大亮点，它与 java.io 包里的 InputStream 和 OutputStream 是完全不相关的东西。

Java 8 中的 Stream 是对集合（Collection）对象功能的增强，它专注于对集合对象进行各种非常便利、高效的聚合操作（aggregate operation），或者大批量数据操作 (bulk data operation)。

Java 8 中出现的 java.util.stream 是一个函数式语言+多核时代综合影响的产物。

这里一个简单的示例——对数组求和。

在引入流之前：

```
int[] nums = { 1, 2, 3 };
//循环计算求职
int sum = 0;
for (int i : nums) {
    sum += i;
}
System.out.println("结果为：" + sum);
```

逻辑也比较简单，引入流之后：

```
 //使用流
int sum2 = IntStream.of(nums).sum();
System.out.println("结果为：" + sum2);
```

代码相对而言要简洁一些。

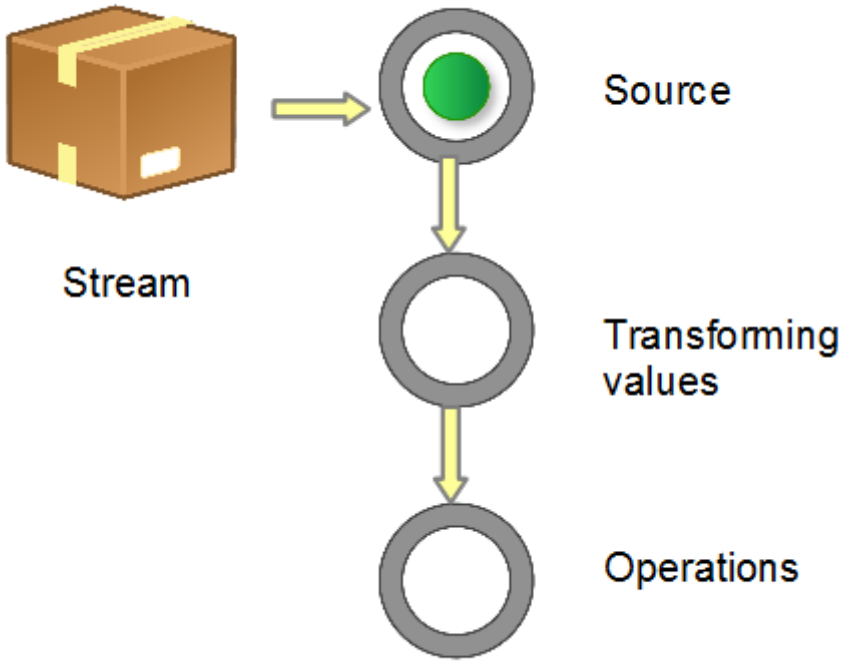
这只是一个简单的迭代求和，如果是一些复杂的聚合或批量操作，那么流在代码简洁性上就更有优势了。

2、创建流

当我们使用一个流的时候，通常包括三个基本步骤：

获取一个数据源（source）→ 数据转换→ 执行操作获取想要的结果。

每次转换原有 Stream 对象不改变，返回一个新的 Stream 对象（可以有多次转换），这就允许对其操作可以像链条一样排列，变成一个管道，如下图所示。



有很多方法可以创建不同数据源的流实例。

2.1、空的流

创建空的流，使用empty()方法：

```
Stream<String> streamEmpty = Stream.empty();
```

使用empty()方法创建来避免没有元素的流返回null的问题：

```
    public Stream<String> streamOf(List<String> list) {
        return list == null || list.isEmpty() ? Stream.empty() : list.stream();
    }
```

2.2、集合的流

可以创建任何类型的集合（Collection, List, Set）的流：

```
Collection<String> collection = Arrays.asList("a", "b", "c");
    Stream<String> streamOfCollection = collection.stream();
```

2.3、数组的流

数组也可以作为流的数据源：

```
Stream<String> streamOfArray = Stream.of("a", "b", "c");
```

也可以从现有数组或数组的一部分中创建流：

```
    String[] arr = new String[]{"a", "b", "c"};
    Stream<String> streamOfArrayFull = Arrays.stream(arr);
    Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

2.4、 Stream.builder()

使用builder时，应在语句的右侧另外使用的类型，否则build()方法将创建 Stream <Object >的实例：

```
Stream<String> streamBuilder =
      Stream.<String>builder().add("a").add("b").add("c").build();
```

2.5、 Stream.generate()

generate()方法接受Supplier\ 进行元素生成。由于结果流是无限的，因此开发人员应指定所需的大小，否则generate()方法运行后会达到内存的上限：

```
Stream<String> streamGenerated =
      Stream.generate(() -> "element").limit(10);
```

上面的代码创建了一个由十个字符串组成的序列，其值是“ element”。

2.6、 Stream.iterate()

创建无限流的另一种方法是使用iterate()方法：

```
Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
```

结果流的第一个元素是iterate()方法的第一个参数。为了创建后续的元素，使用了上一个元素。在上面的示例中，第二个元素为42。

2.7、 基本类型的流

Java 8提供了从三种基本类型中创建流的方式：int，long和double。

由于Stream <T>是泛型接口，无法将基本类型用作泛型的类型参数，因此创建了三个新的特殊接口：IntStream，LongStream和DoubleStream。使用新接口可以减轻不必要的自动装箱，从而提效率：

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
```

range（int startInclusive，int endExclusive）方法创建从第一个参数到第二个参数的有序流。它以等于1的步长递增后续元素的值。结果不包括最后一个参数，它只是序列的上限。

2.8、字符串的流

字符串也可以用作创建流的数据源。

由于JDK中没有接口CharStream，因此使用IntStream表示字符流。用到了String类的chars()方法。

```
IntStream streamOfChars = "abc".chars();
```

下面的示例根据指定的正则表达式将String细分为子字符串：

```
Stream<String> streamOfString =
  Pattern.compile(", ").splitAsStream("a, b, c");
```

2.9、文件的流

Java NIO类 Files 允许通过lines()方法生成文本文件的Stream <String>。文本的每一行都成为流的元素：

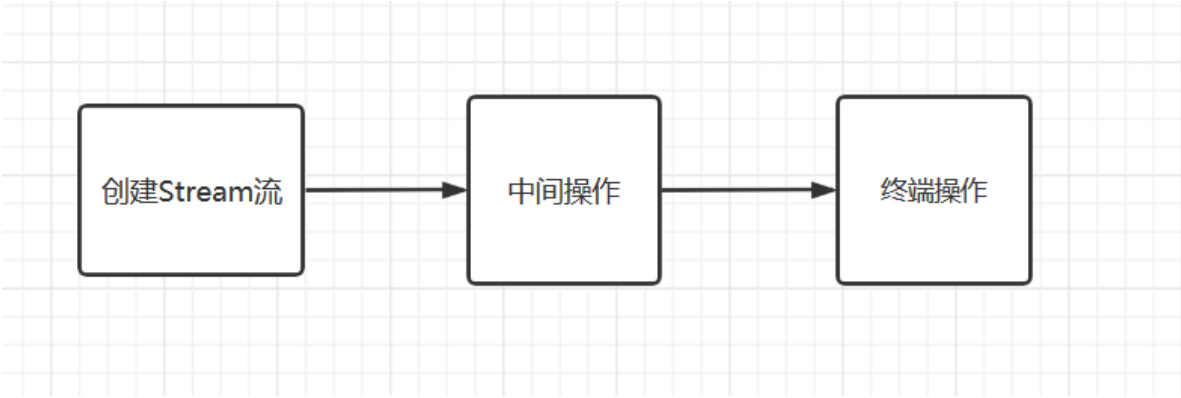
```
Path path = Paths.get("C:\\file.txt");
    Stream<String> streamOfStrings = Files.lines(path);
    Stream<String> streamWithCharset =
      Files.lines(path, Charset.forName("UTF-8"));
```

可以将字符编码指定为lines()方法的参数。

最常用的就是从集合中创建出流。

3、流操作

上面学习了流的创建方式，接下来学习流的操作。



它们分为中间操作（返回Stream <T>）和终端操作（返回确定类型的结果）。中间操作允许链式传递。

我们来看一看常用的非终端操作。

3.1、中间操作

中间操作也叫非终端操作。

Java Stream API的非终端流操作是对流中的元素进行转换或过滤的操作。

当向流添加非终端操作时，将得到一个新的流。新流表示应用了非终端操作的原始流产生的元素流。这是添加到流中的非终端操作的示例——会产生新的流：

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

stringList.add("ONE");
stringList.add("TWO");
stringList.add("THREE");

Stream<String> stream = stringList.stream();

Stream<String> stringStream =
    stream.map((value) -> { return value.toLowerCase(); });
```

注意对stream.map()的调用。该调用实际上返回一个新的Stream实例，该实例表示已使用map操作原来的字符流。

只能将单个操作添加到给定的Stream实例。

如果需要将多个操作彼此链接在一起，则需要将第二个操作应用第一个操作产生的Stream流。实例如下：

```
Stream<String> stringStream1 =
        stream.map((value) -> { return value.toLowerCase(); });

Stream<½String> stringStream2 =
        stringStream1.map((value) -> { return value.toUpperCase(); });
```

链式调用是非常常见的，这是链式调用的实例：

```
Stream<String> stream1 = stream
  .map((value) -> { return value.toLowerCase(); })
  .map((value) -> { return value.toUpperCase(); })
  .map((value) -> { return value.substring(0,3); });
```

许多非终端Stream操作可以将Java Lambda表达式作为参数。该lambda表达式实现了适合给定非终端操作的Java函数式接口。非终端操作方法参数的参数通常是函数式接口——这就是为什么它也可以由Java lambda表达式实现的原因。

3.1.1、filter()

Java Stream filter()可用于过滤Java Stream中的元素。filter方法采用一个Predicate，该Predicate被流中的每个元素被调用。如果元素要包含在结果流中，则Predicate返回true。如果不应包含该元素，则Predicate返回false。

下面是一个filter()实例：

```
Stream<String> longStringsStream = stream.filter((value) -> {
    return value.length() >= 3;
});
```

3.1.2、map()

Java Stream map()方法将一个元素转换（映射）到另一个对象。例如，如果你有一个字符串列表，则可以将每个字符串转换为小写，大写或原始字符串的子字符串，或者完全转换成其他字符串。

这是一个Java Stream map()示例：

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

Stream<String> streamMapped = stream.map((value) -> value.toUpperCase());
```

3.1.3、flatMap()

Java Stream flatMap()方法将单个元素映射到多个元素。意思是将每个元素从由多个内部元素组成的复杂结构“展平”到仅由这些内部元素组成的“展平”流。

例如，假设你有一个带有嵌套对象（子对象）的对象。然后，你可以将该对象映射到一个“平”流，该流由自身加上其嵌套对象——或仅嵌套对象组成。你还可以将元素列表流映射到元素本身。或将字符串流映射到这些字符串中的字符流——或映射到这些字符串中的各个Character实例。

这是一个将字符串列表平面映射到每个字符串中的字符的示例。

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

stringList.add("One flew over the cuckoo's nest");
stringList.add("To kill a muckingbird");
stringList.add("Gone with the wind");

Stream<String> stream = stringList.stream();

stream.flatMap((value) -> {
    String[] split = value.split(" ");
    return (Stream<String>) Arrays.asList(split).stream();
}).forEach((value) -> System.out.println(value));
```

此Java Stream flatMap() 示例首先创建一个包含3个包含书名的字符串的List。然后，获取List的Stream，并调用flatMap()。

3.1.4、distinct()

Java Stream unique()方法是一种非终端操作，返回一个新的Stream，与来源的流不同，它去掉了重复的元素。这是Java Streamdisting()方法的示例：

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

        stringList.add("one");
        stringList.add("two");
        stringList.add("three");
        stringList.add("one");

        Stream<String> stream = stringList.stream();

        List<String> distinctStrings = stream
                .distinct()
                .collect(Collectors.toList());

        System.out.println(distinctStrings);
```

在此示例中，元素 " one " 在原来的流中出现2次。在新的流中只会出现第一次出现的元素。因此，结果列表（通过调用collect() 将仅包含 "one"，"two" 和"three"。从此示例打印的输出将是：

```
[one, two, three]
```

3.1.5、limit()

Java Stream limit()方法可以将流中的元素数量限制为指定给limit()方法的数量。limit() 方法返回一个新的Stream，该Stream最多包含给定数量的元素。这是一个Java Stream limit() 示例：

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

        stringList.add("one");
        stringList.add("two");
        stringList.add("three");
        stringList.add("one");

        Stream<String> stream = stringList.stream();
        stream
            .limit(2)
            .forEach( element -> { System.out.println(element); });
```

本示例首先创建一个Stream流，然后在其上调用limit()，然后使用forEach()来打印出该流中的元素。由于调用了limit(2)，仅将打印前两个元素。

3.1.6、peek()

Java Stream peek()方法是一种非终端操作，它以 Consumer（java.util.function.Consumer）作为参数。将为流中的每个元素调用Consumer。peek()方法返回一个新的Stream，其中包含原来的流中的所有元素。

正如方法所说，peek() 方法的目的是见识流中的元素，而不是对其进行转换。peek方法不会启动流中元素的内部迭代。要这是一个Java Stream peek()示例：

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

stringList.add("abc");
stringList.add("def");

Stream<String> stream = stringList.stream();

Stream<String> streamPeeked = stream.peek((value) -> {
    System.out.println("value");
});
```

3.2、终端操作

Java Stream接口的终端操作通常返回单个值。一旦在Stream上调用了终端操作，就将开始Stream的迭代以及链路上的流。迭代完成后，将返回终端操作的结果。

终端操作通常不返回新的Stream实例。因此，一旦在流上调用了终端操作，来自非终端操作的Stream实例链就结束了。

这是在Java Stream上调用终端操作的示例：

```
long count = stream
  .map((value) -> { return value.toLowerCase(); })
  .map((value) -> { return value.toUpperCase(); })
  .map((value) -> { return value.substring(0,3); })
  .count();
```

该示例末尾的对count()的调用是终端操作。由于count()返回long，因此非终端操作的Stream链路（map()调用）结束。

3.2.1、anyMatch()

Java Stream anyMatch()方法是一种终端操作，它以单个Predicate作为参数，启动Stream的内部迭代，并将Predicate参数应用于每个元素。如果Predicate对任意一个元素返回true，则anyMatch（）方法返回true。如果没有元素与Predicate匹配，则anyMatch()将返回false。

这是一个Java Stream anyMatch（）示例：

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

        stringList.add("One flew over the cuckoo's nest");
        stringList.add("To kill a muckingbird");
        stringList.add("Gone with the wind");

        Stream<String> stream = stringList.stream();

        boolean anyMatch = stream.anyMatch((value) -> { return
value.startsWith("One"); });
        System.out.println(anyMatch);
```

在上面的示例中，anyMatch() 方法调用将返回true，因为流中的第一个字符串元素以“ One”开头。

3.2.2、allMatch()

Java Stream allMatch() 方法是一种终端操作，该操作以单个Predicate作为参数，启动Stream中元素的内部迭代，并将Predicate参数应用于每个元素。如果Predicate对于Stream中的所有元素都返回true，则allMatch() 将返回true。如果不是所有元素都与Predicate匹配，则allMatch() 方法将返回false。

这是一个Java Stream allMatch() 示例：

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

stringList.add("One flew over the cuckoo's nest");
stringList.add("To kill a muckingbird");
stringList.add("Gone with the wind");

Stream<String> stream = stringList.stream();

boolean allMatch = stream.allMatch((value) -> { return value.startsWith("One");
});
System.out.println(allMatch);
```

在上面的示例中，allMatch()方法将返回false，因为Stream中只有一个字符串以“ One”开头。

3.2.3、noneMatch()

Java Stream noneMatch() 方法是一个终端操作，它将对流中的元素进行迭代并返回true或false，这取决于流中是否没有元素与作为参数传递给noneMatch() 的谓词相匹配。如果谓词不匹配任何元素，则noneMatch() 方法将返回true；如果匹配一个或多个元素，则方法将返回false。

这是一个Java Stream noneMatch() 示例：

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

stringList.add("abc");
stringList.add("def");

Stream<String> stream = stringList.stream();

boolean noneMatch = stream.noneMatch((element) -> {
    return "xyz".equals(element);
});

System.out.println("noneMatch = " + noneMatch);
```

3.2.4、collect()

Java Stream collect() 方法是一种终端操作，它开始元素的内部迭代，并以某种类型的集合或对象接收流中的元素。

这是一个简单的Java Stream collect()方法示例：

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

stringList.add("One flew over the cuckoo's nest");
stringList.add("To kill a muckingbird");
stringList.add("Gone with the wind");

Stream<String> stream = stringList.stream();

List<String> stringsAsUppercaseList = stream
.map(value -> value.toUpperCase())
.collect(Collectors.toList());

System.out.println(stringsAsUppercaseList);
```

collect() 方法采用Collector（java.util.stream.Collector）作为参数。实现Collector需要对Collector接口进行一些研究。幸运的是，Java类java.util.stream.Collectors包含了一组可以用于最常用操作的预先实现的Collector实现。在上面的示例中，使用的是Collectors.toList() 返回的Collector实现。该Collector只是将流中的所有元素收集到标准Java List中。

3.2.5、 count()

Java Stream count() 方法是一种终端操作，用于启动Stream中元素的内部迭代并计算元素。这是一个Java Stream count() 示例：

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

stringList.add("One flew over the cuckoo's nest");
stringList.add("To kill a muckingbird");
stringList.add("Gone with the wind");

Stream<String> stream = stringList.stream();

long count = stream.flatMap((value) -> {
    String[] split = value.split(" ");
    return (Stream<String>) Arrays.asList(split).stream();
})
.count();

System.out.println("count = " + count);
```

此示例首先创建一个字符串列表，然后获取该列表的Stream，为其添加一个flatMap()操作，然后完成对count()的调用。count()方法将开始Stream中元素的迭代，flatMap()操作中将字符串元素拆分为单词，然后进行计数。最终打印出来的结果是14。

3.2.6、findAny()

Java Stream findAny() 方法可以从Stream中查找单个元素。找到的元素可以来自Stream中的任何位置。无法保证从流中何处获取元素。

这是一个Java Stream findAny()示例：

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

stringList.add("one");
stringList.add("two");
stringList.add("three");
stringList.add("one");

Stream<String> stream = stringList.stream();

Optional<String> anyElement = stream.findAny();

System.out.println(anyElement.get());
```

注意findAny()方法返回了Optional。Stream可能为空——因此无法返回任何元素。可以检查是否通过可选的isPresent()方法找到元素。

3.2.7、findFirst()

如果Stream中存在任何元素，则Java Stream findFirst（）方法将查找Stream中的第一个元素。findFirst（）方法返回一个Optional，可以从中获取元素（如果存在）。

这是一个Java Stream findFirst() 示例：

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

stringList.add("one");
stringList.add("two");
stringList.add("three");
stringList.add("one");

Stream<String> stream = stringList.stream();

Optional<String> result = stream.findFirst();

System.out.println(result.get());
```

可以通过isPresent() 方法检查Optional返回的元素是否包含元素。

3.2.8、forEach()

Java Stream forEach() 方法是一种终端操作，它对Stream中元素迭代，并将Consumer（java.util.function.Consumer）应用于Stream中的每个元素。forEach() 无返回值。

这是一个Java Stream forEach() 示例：

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

        stringList.add("one");
        stringList.add("two");
        stringList.add("three");
        stringList.add("one");

        Stream<String> stream = stringList.stream();

        stream.forEach( element -> { System.out.println(element); });
```

3.2.9、min()

Java Stream min() 方法是一种终端操作，它返回Stream中的最小元素。哪个元素最小是由传递给min() 方法的Comparator实现确定的。

这是一个Java Stream min() 示例：

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

stringList.add("abc");
stringList.add("def");

Stream<String> stream = stringList.stream();

Optional<String> min = stream.min((val1, val2) -> {
    return val1.compareTo(val2);
});

String minString = min.get();

System.out.println(minString);
```

注意min() 方法返回一个Optional，它可能包含也可能不包含结果。如果Stream为空，则Optional get()方法将抛出NoSuchElementException。

3.2.10、max()

Java Stream max() 方法是一种终端操作，它返回Stream中最大的元素。哪个元素最大，取决于传递给max() 方法的Comparator实现。

这是一个Java Stream max() 示例：

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

stringList.add("abc");
stringList.add("def");

Stream<String> stream = stringList.stream();

Optional<String> max = stream.max((val1, val2) -> {
    return val1.compareTo(val2);
});

String maxString = max.get();

System.out.println(maxString);
```

注意max() 方法如何返回一个Optional，它可以包含也可以不包含结果。如果Stream为空，则Optional get（）方法将抛出NoSuchElementException。

3.2.11、reduce()

Java Stream reduce() 方法是一种终端操作，可以将流中的所有元素缩减为单个元素。

这是一个Java Stream reduce() 示例：

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

stringList.add("One flew over the cuckoo's nest");
stringList.add("To kill a muckingbird");
stringList.add("Gone with the wind");

Stream<String> stream = stringList.stream();

Optional<String> reduced = stream.reduce((value, combinedValue) -> {
    return combinedValue + " + " + value;
});

System.out.println(reduced.get());
```

3.2.12、toArray()

Java Stream toArray() 方法是一种终端操作，它迭代流中元素，并返回包含所有元素的Object数组。

这是一个Java Stream toArray() 示例：

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

stringList.add("One flew over the cuckoo's nest");
stringList.add("To kill a muckingbird");
stringList.add("Gone with the wind");

Stream<String> stream = stringList.stream();

Object[] objects = stream.toArray();
```

3.2.13、sorted()

Collection 需要排序的时候可以使用Comparator和Comparable实现。在Java 8中，同样可以使用Comparator对Stream进行排序。

示例如下：

```
public class Human {
    private String name;
    private int age;
}
```

```
        ArrayList<Human> humans = new ArrayList<Human>();
        humans.add(new Human("李四", 4));
        humans.add(new Human("王二", 2));
        humans.add(new Human("张三", 3));
        System.out.println(humans);
        List<Human> sortHumans =
humans.stream().sorted(Comparator.comparing(Human::getAge))
                .collect(Collectors.toList());
        System.out.println(sortHumans);
```

4、总结

Stream API是一组功能强大但易于理解的工具，用于处理元素序列。如果使用得当，我们可以减少大量的重复代码，创建更具可读性的程序，并提高应用的工作效率。

参考：

【1】：Java 8 中的 Streams API 详解
【2】：[译] 一文带你玩转 Java8 Stream 流，从此操作集合 So Easy
【3】：A Guide to Streams in Java 8: In-Depth Tutorial With Examples
【4】：The Java 8 Stream API Tutorial
【5】：java.util.stream
【6】：Introduction to Java 8 Streams
【7】：Java Stream API
【8】：Java8 使用 stream().sorted()对List集合进行排序
【9】：Java 8 Stream sorted() Example