Java 8 新特性 之 Stream 流

作者：wubo8196
原文链接：https://ld246.com/article/1576228001270
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Stream 流

说到 Stream 便容易想到 I/O Stream，而实际上，谁规定“流”就一定是“IO 流”呢？在 Java 8 ，得益于 Lambda 所带 来的函数式编程，引入了一个全新的 Stream 概念，用于解决已有集合类库有的弊端。

1.1 引言

传统集合的多步遍历代码

几乎所有的集合（如 Collection 接口或 Map 接口等）都支持直接或间接的遍历操作。而当我们需要集合中的元素进行操作的时候，除了必需的添加、删除、获取外，最典型的就是集合遍历。例如：

```
import java.util.ArrayList;
import java.util.List;
public class Demo01ForEach {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");
        for (String name : list) {
            System.out.println(name);
        }
    }
}
```

这是一段非常简单的集合遍历操作：对集合中的每一个字符串都进行打印输出操作。

循环遍历的弊端

Java 8 的 Lambda 让我们可以更加专注于**做什么**(What)，而不是**怎么做**(How)，这点此前已经结合部类进行了对比说明。现在，我们仔细体会一下上例代码，可以发现：

- for 循环的语法就是 **怎么做**
- for 循环的循环体才是 **做什么**

为什么使用循环？因为要进行遍历。但循环是遍历的唯一方式吗？遍历是指每一个元素逐一进行处理，**并不是从第一个到最后一个顺次处理的循环**。前者是目的，后者是方式。

试想一下，如果希望对集合中的元素进行筛选过滤：

1. 将集合 A 根据条件一过滤为 **子集 B**。
2. 然后再根据条件二过滤为 **子集 C**。

那怎么办？在 Java 8 之前的做法可能为：

```
import java.util.ArrayList;
import java.util.List;
public class Demo02NormalFilter {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");
        List<String> zhangList = new ArrayList<>();
        for (String name : list) {
            if (name.startsWith("张")) {
                zhangList.add(name);
            }
        }
        List<String> shortList = new ArrayList<>();
        for (String name : zhangList) {
            if (name.length() == 3) {
                shortList.add(name);
            }
        }
        for (String name : shortList) {
            System.out.println(name);
        }
    }
}
```

这段代码中含有三个循环，每一个作用不同：

1. 首先筛选所有姓张的人；
2. 然后筛选名字有三个字的人；

3. 最后进行对结果进行打印输出。

每当我们需要对集合中的元素进行操作的时候，总是需要进行循环、循环、再循环。这是理所当然的？**不是**。循环是做事情的方式，而不是目的。另一方面，使用线性循环就意味着只能遍历一次。如果望再次遍历，只能再使用另一个循环从头开始。

那，Lambda 的衍生物 Stream 能给我们带来怎样更加优雅的写法呢？

Stream 的更优写法

借助 Java 8 的 Stream API，上面需求可以用更优雅的方式完成：

```
import java.util.ArrayList;
import java.util.List;
public class Demo03StreamFilter {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");
        list.stream()
            .filter(s -> s.startsWith("张"))
            .filter(s -> s.length() == 3)
            .forEach(System.out::println);
    }
}
```

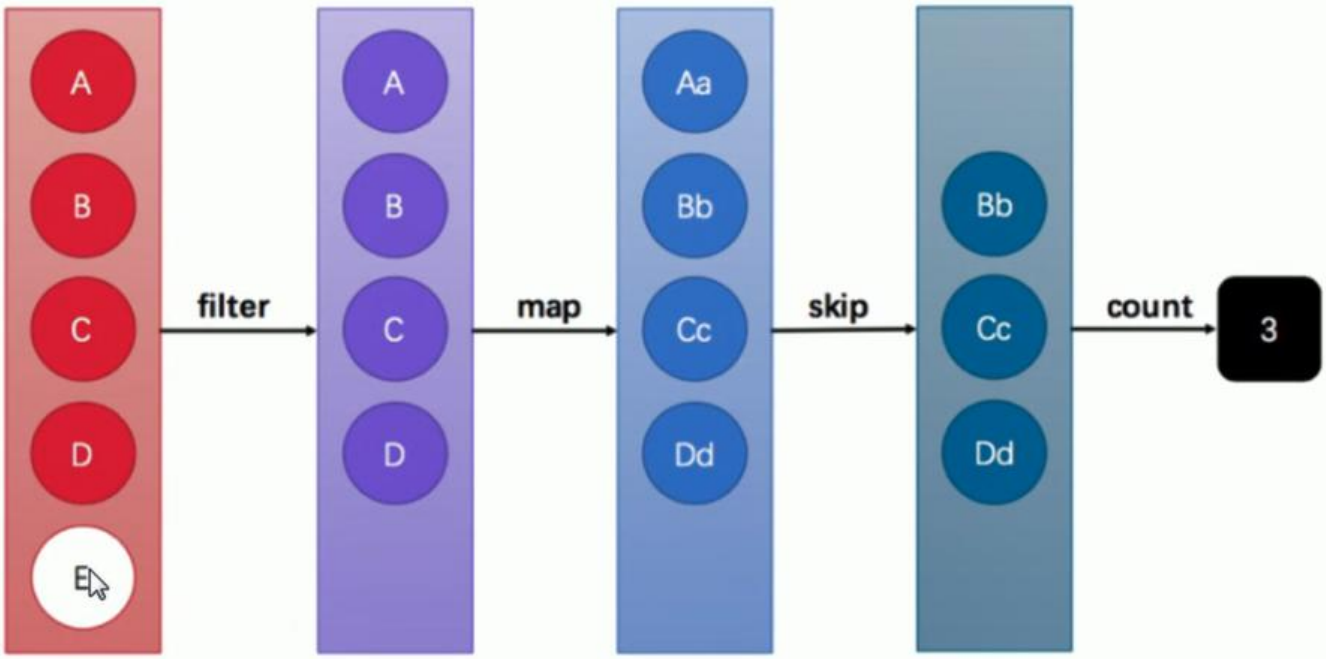
直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：**获取流**、**过滤姓张**、**过滤长度为 3**、**逐打印**。代码中并没有体现使用线性循环或是其他任何算法进行遍历，我们真正要做的事情内容被更好体现在代码中。

1.2 流式思想概述

流式思想不同于传统 IO 流

整体来看，流式思想类似于工厂车间的**生产流水线**。

当需要对多个元素进行操作（特别是多步操作）的时候，考虑到性能及便利性，我们应该首先拼好一“模型”步骤方案，然后再按照方案去执行它。



这张图中展示了过滤、映射、跳过、计数等多步操作，这是一种集合元素的处理方案，而方案就是一“函数模型”。图中的每一个方框都是一个“流”，调用指定的方法，可以从一个流模型转换为另一流模型。而最右侧的数字 3 是最终结果。

这里的 filter 、 map 、 skip 都是在对函数模型进行操作，集合元素并没有真正被处理。只有当终结法 count 执行的时候，整个模型才会按照指定策略执行操作。而这得益于 Lambda 的延迟执行特性。

备注：“Stream 流”其实是一个集合元素的函数模型，它并不是集合，也不是数据结构，其本身并存储任何元素（或其地址值）。

Stream（流）是一个来自数据源的元素队列

- 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 entstyle）。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 或者增强 for 的方式， 显式的在集合外部进行迭代，这叫做外部迭代。Stream 提供了内部迭代的方式，流可以直接调用遍历方法。

当使用一个流的时候，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。

1.3 获取流

---

java.util.stream.Stream<T> 是 Java 8 新加入的最常用的流接口。（这并不是一个函数式接口。）

获取一个流非常简单，有以下几种常用的方式：

- 所有的 Collection 集合都可以通过 stream 默认方法获取流；
- Stream 接口的静态方法 of 可以获取数组对应的流。

根据 Collection 获取流

首先， java.util.Collection 接口中加入了 default 方法 sream 用来获取流，所以其所有实现类均可取流。

```
import java.util.*;
import java.util.stream.Stream;
public class Demo04GetStream {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        // ...
        Stream<String> stream1 = list.stream();
        Set<String> set = new HashSet<>();
        // ...
        Stream<String> stream2 = set.stream();
        Vector<String> vector = new Vector<>();
        // ...
        Stream<String> stream3 = vector.stream();
    }
}
```

根据 Map 获取流

java.util.Map 接口不是 Collection 的子接口，且其 K-V 数据结构不符合流元素的单一特征，所以获对应的流

需要分 key、value 或 entry 等情况：

```
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Stream;
public class Demo05GetStream {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        // ...
        Stream<String> keyStream = map.keySet().stream();
        Stream<String> valueStream = map.values().stream();
        Stream<Map.Entry<String, String>> entryStream = map.entrySet().stream();
    }
}
```

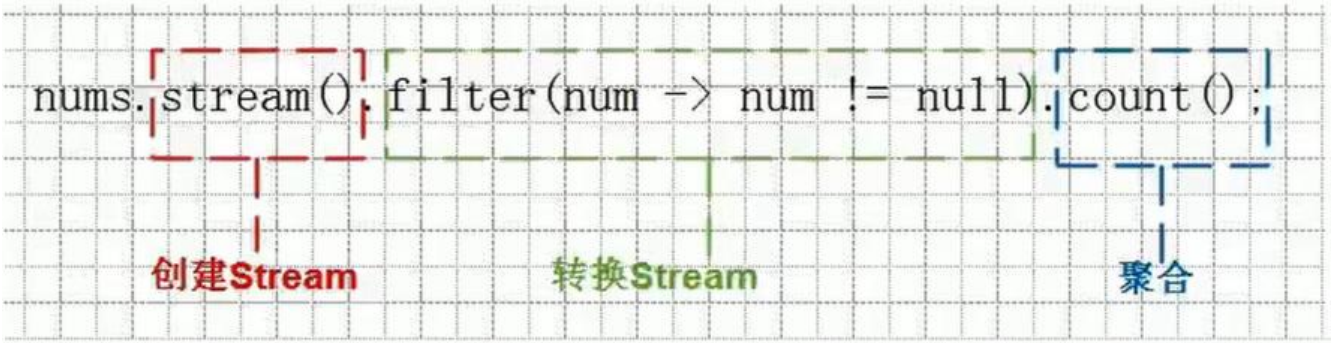
根据数组获取流

如果使用的不是集合或映射而是数组，由于数组对象不可能添加默认方法，所以 Stream 接口中提供静态方法 of ，使用很简单：

```
import java.util.stream.Stream;
public class Demo06GetStream {
    public static void main(String[] args) {
        String[] array = { "张无忌", "张翠山", "张三丰", "张一元" };
        Stream<String> stream = Stream.of(array);
    }
}
```

备注： of 方法的参数其实是一个可变参数，所以支持数组。

1.4 常用方法



流模型的操作很丰富，这里介绍一些常用的 API。这些方法可以被分成两种：

- **延迟方法**：返回值类型仍然是 Stream 接口自身类型的方法，因此支持链式调用。（除了终结方法，其余方法均为延迟方法。）
- **终结方法**：返回值类型不再是 Stream 接口自身类型的方法，因此不再支持类似 StringBuilder 那的链式调用。本小节中，终结方法包括 count 和 forEach 方法。

逐一处理：forEach

虽然方法名字叫 forEach ，但是与 for 循环中的“for-each”昵称不同。

```
void forEach(Consumer<? super T> action);
```

该方法接收一个 Consumer 接口函数，会将每一个流元素交给该函数进行处理。

java.util.function.Consumer<T> 接口是一个消费型接口。 Consumer 接口中包含抽象方法 void accept(T t)，意为消费一个指定泛型的数据。

基本使用：

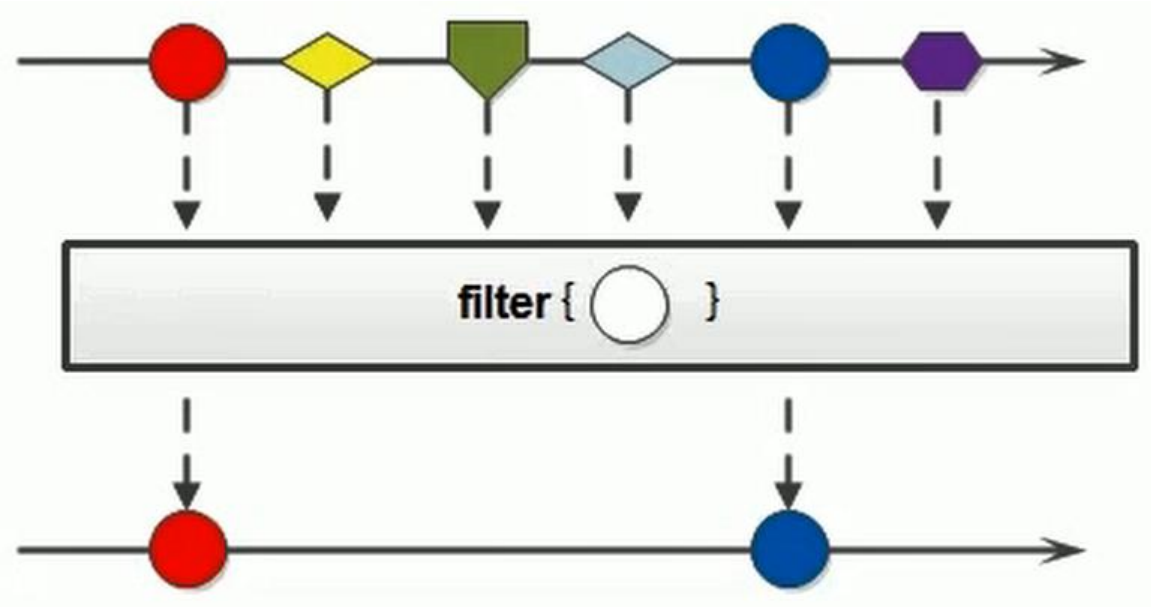
```
import java.util.stream.Stream;
public class Demo12StreamForEach {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("张无忌", "张三丰", "周芷若");
        stream.forEach(name-> System.out.println(name));
    }
}
```

过滤：filter

可以通过 filter 方法将一个流转换成另一个子集流。方法签名：

```
Stream<T> filter(Predicate<? super T> predicate);
```

该接口接收一个 Predicate 函数式接口参数（可以是一个 Lambda 或方法引用）作为筛选条件。



java.util.stream.Predicate 函数式接口，其中唯一的抽象方法为：

```
boolean test(T t);
```

该方法将会产生一个 boolean 值结果，代表指定的条件是否满足。如果结果为 true，那么 Stream 的 filter 方法将会留用元素；如果结果为 false，那么 filter 方法将会舍弃元素。

基本使用：

Stream 流中的 filter 方法基本使用的代码如：

```
import java.util.stream.Stream;
public class Demo07StreamFilter {
```

```
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.filter(s -> s.startsWith("张"));
    }
}
```

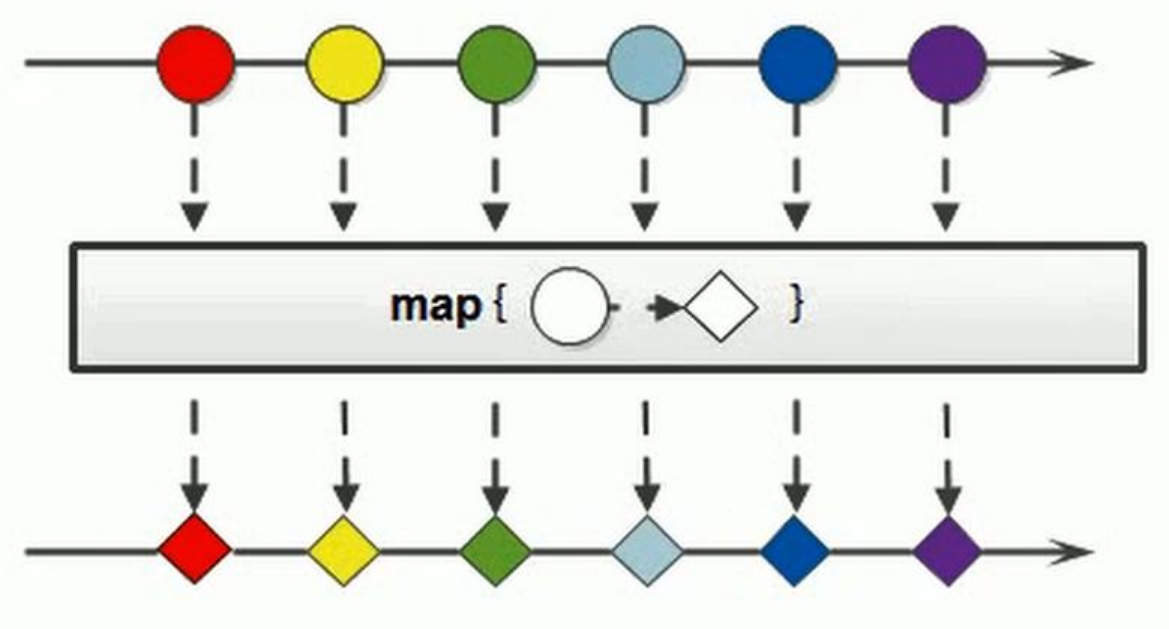
在这里通过 Lambda 表达式来指定了筛选的条件：必须姓张。

映射：map

如果需要将流中的元素映射到另一个流中，可以使用 map 方法。方法签名：

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

该接口需要一个 Function 函数式接口参数，可以将当前流中的 T 类型数据转换为另一种 R 类型的流。



java.util.stream.Function 函数式接口，其中唯一的抽象方法为：

```
R apply(T t);
```

这可以将一种 T 类型转换成为 R 类型，而这种转换的动作，就称为“映射”。

基本使用：

Stream 流中的 map 方法基本使用的代码如：

```
import java.util.stream.Stream;
public class Demo08StreamMap {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("10", "12", "18");
        Stream<Integer> result = original.map(str->Integer.parseInt(str));
    }
}
```

这段代码中， map 方法的参数通过方法引用，将字符串类型转换成为了 int 类型（并自动装箱为 Integer 类对

象）。

统计个数： count

正如旧集合 Collection 当中的 size 方法一样，流提供 count 方法来数一数其中的元素个数：

```
long count();
```

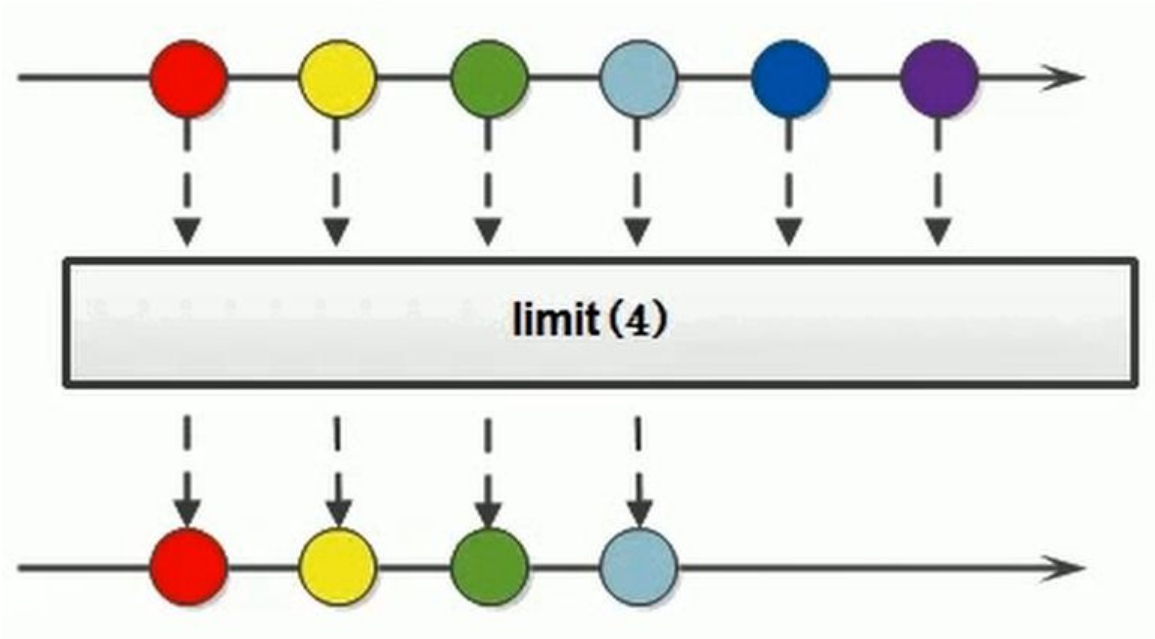
该方法返回一个 long 值代表元素个数（不再像旧集合那样是 int 值）。基本使用：

```
import java.util.stream.Stream;
public class Demo09StreamCount {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.filter(s -> s.startsWith("张"));
        System.out.println(result.count()); // 2
    }
}
```

取用前几个：limit

limit 方法可以对流进行截取，只取用前 n 个。方法签名：

```
Stream<T> limit(long maxSize);
```



参数是一个 long 型，如果集合当前长度大于参数则进行截取；否则不进行操作。基本使用：

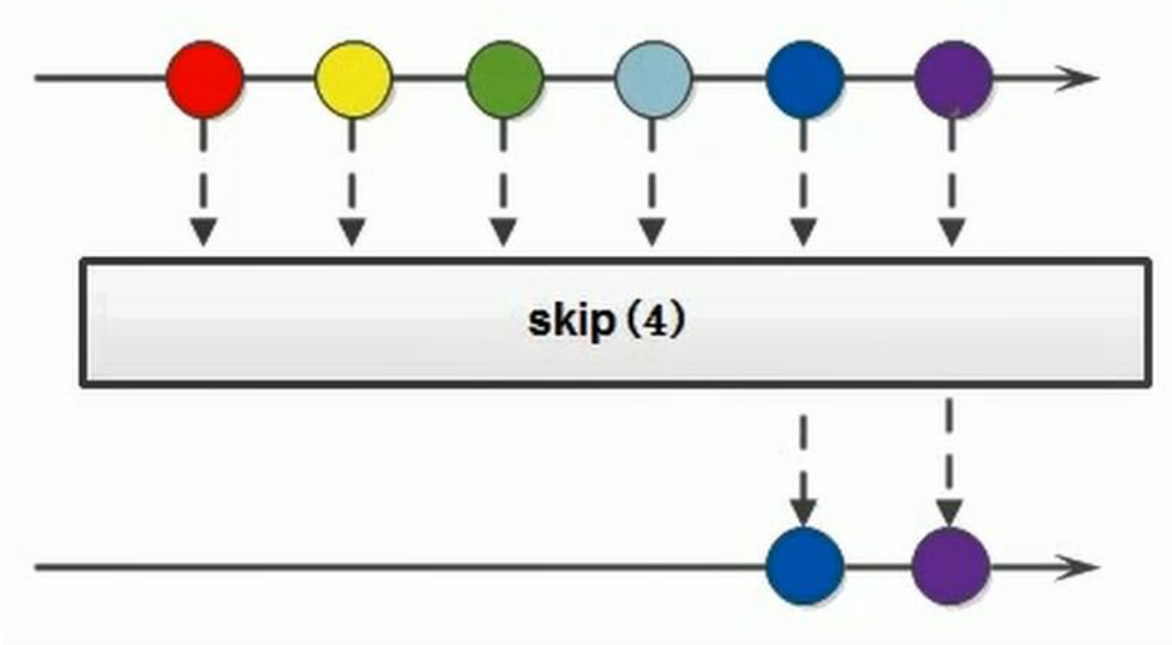
```
import java.util.stream.Stream;
public class Demo10StreamLimit {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.limit(2);
        System.out.println(result.count()); // 2
    }
}
```

跳过前几个：skip

如果希望跳过前几个元素，可以使用 skip 方法获取一个截取之后的新流：

```
Stream<T> skip(long n);
```

如果流的当前长度大于 n，则跳过前 n 个；否则将会得到一个长度为 0 的空流。基本使用：




```
import java.util.stream.Stream;
public class Demo11StreamSkip {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.skip(2);
        System.out.println(result.count()); // 1
    }
}
```

组合：concat

如果有两个流，希望合并成为一个流，那么可以使用 Stream 接口的静态方法 concat ：

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

备注：这是一个静态方法，与 java.lang.String 当中的 concat 方法是不同的。

该方法的基本使用代码如：

```
import java.util.stream.Stream;
public class Demo12StreamConcat {
    public static void main(String[] args) {
        Stream<String> streamA = Stream.of("张无忌");
        Stream<String> streamB = Stream.of("张翠山");
        Stream<String> result = Stream.concat(streamA, streamB);
    }
}
```

1.5 练习：集合元素处理（传统方式）

题目

现在有两个 ArrayList 集合存储队伍当中的多个成员姓名，要求使用传统的 for 循环（或增强 for 循）依次进行以下若干操作步骤：

1. 第一个队伍只要名字为 3 个字的成员姓名；存储到一个新集合中。
2. 第一个队伍筛选之后只要前 3 个人；存储到一个新集合中。
3. 第二个队伍只要姓张的成员姓名；存储到一个新集合中。
4. 第二个队伍筛选之后不要前 2 个人；存储到一个新集合中。
5. 将两个队伍合并为一个队伍；存储到一个新集合中。
6. 根据姓名创建 Person 对象；存储到一个新集合中。
7. 打印整个队伍的 Person 对象信息。

两个队伍（集合）的代码如下：

```
import java.util.ArrayList;
import java.util.List;
public class DemoArrayListNames {
    public static void main(String[] args) {
        //第一支队伍
        ArrayList<String> one = new ArrayList<>();
        one.add("迪丽热巴");
        one.add("宋远桥");
        one.add("苏星河");
        one.add("石破天");
        one.add("石中玉");
        one.add("老子");
        one.add("庄子");
        one.add("洪七公");
        //第二支队伍
        ArrayList<String> two = new ArrayList<>();
        two.add("古力娜扎");
        two.add("张无忌");
        two.add("赵丽颖");
        two.add("张三丰");
        two.add("尼古拉斯赵四");
        two.add("张天爱");
        two.add("张二狗");
        // ....
    }
}
```

而 Person 类的代码为：

```
public class Person {
    private String name;
    public Person() {}
    public Person(String name) {
```

```
        this.name = name;
    }
    @Override
    public String toString() {
        return "Person{name='" + name + "'}";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

解答

既然使用传统的 for 循环写法，那么：

```
public class DemoArrayListNames {
    public static void main(String[] args) {
        List<String> one = new ArrayList<>();
        // ...
        List<String> two = new ArrayList<>();
        // ...
        // 第一个队伍只要名字为3个字的成员姓名；
        List<String> oneA = new ArrayList<>();
        for (String name : one) {
            if (name.length() == 3) {
                oneA.add(name);
            }
        }
        // 第一个队伍筛选之后只要前3个人；
        List<String> oneB = new ArrayList<>();
        for (int i = 0; i < 3; i++) {
            oneB.add(oneA.get(i));
        }
        // 第二个队伍只要姓张的成员姓名；
        List<String> twoA = new ArrayList<>();
        for (String name : two) {
            if (name.startsWith("张")) {
                twoA.add(name);
            }
        }
        // 第二个队伍筛选之后不要前2个人；
        List<String> twoB = new ArrayList<>();
        for (int i = 2; i < twoA.size(); i++) {
            twoB.add(twoA.get(i));
        }
        // 将两个队伍合并为一个队伍；
        List<String> totalNames = new ArrayList<>();
        totalNames.addAll(oneB);
        totalNames.addAll(twoB);
        // 根据姓名创建Person对象；
        List<Person> totalPersonList = new ArrayList<>();
        for (String name : totalNames) {
```

```
            totalPersonList.add(new Person(name));
        }
        // 打印整个队伍的Person对象信息。
        for (Person person : totalPersonList) {
            System.out.println(person);
        }
    }
}
```

运行结果为：

```
Person{name='宋远桥'}
Person{name='苏星河'}
Person{name='石破天'}
Person{name='张天爱'}
Person{name='张二狗'}
```

1.6 练习：集合元素处理（Stream 方式）

题目

将上一题当中的传统 for 循环写法更换为 Stream 流式处理方式。两个集合的初始内容不变， Person 类的定义也不变。

解答

等效的 Stream 流式处理代码为：

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
public class DemoStreamNames {
    public static void main(String[] args) {
        List<String> one = new ArrayList<>();
        // ...
        List<String> two = new ArrayList<>();
        // ...
        // 第一个队伍只要名字为3个字的成员姓名；
        // 第一个队伍筛选之后只要前3个人；
        Stream<String> streamOne = one.stream().filter(s -> s.length() == 3).limit(3);
        // 第二个队伍只要姓张的成员姓名；
        // 第二个队伍筛选之后不要前2个人；
        Stream<String> streamTwo = two.stream().filter(s -> s.startsWith("张")).skip(2);
        // 将两个队伍合并为一个队伍；
        // 根据姓名创建Person对象；
        // 打印整个队伍的Person对象信息。
        Stream.concat(streamOne, streamTwo).map(Person::new).forEach(System.out::println);
    }
}
```

运行效果完全一样：

```
Person{name='宋远桥'}
```

```
Person{name='苏星河'}
Person{name='石破天'}
Person{name='张天爱'}
Person{name='张二狗'}
```