day14【Stream流】

主要内容

- 常用函数式接口
- Stream流

教学目标

- [ ] 能够使用Function函数式接口
- [ ] 能够使用Predicate函数式接口
- [ ] 能够理解流与集合相比的优点
- [ ] 能够理解流的延迟执行特点
- [ ] 能够通过集合、映射或数组获取流
- [ ] 能够掌握常用的流操作
- [ ] 能够使用流进行并发操作
- [ ] 能够将流中的内容收集到集合中
- [ ] 能够将流中的内容收集到数组中

第一章 常用函数式接口

1.1 Predicate接口

有时候我们需要对某种类型的数据进行判断，从而得到一个boolean值结果。这时可以使用`java.util.function.Predicate<T>`接口。

抽象方法：test

`Predicate`接口中包含一个抽象方法：`boolean test(T t)`。用于条件判断的场景：

```
import java.util.function.Predicate;

public class Demo15PredicateTest {
    private static void method(Predicate<String> predicate) {
        boolean veryLong = predicate.test("HelloWorld");
        System.out.println("字符串很长吗：" + veryLong);
    }

    public static void main(String[] args) {
        method(s -> s.length() > 5);
    }
}
```

条件判断的标准是传入的Lambda表达式逻辑，只要字符串长度大于5则认为很长。

默认方法：and

既然是条件判断，就会存在与、或、非三种常见的逻辑关系。其中将两个`Predicate`条件使用“与”逻辑连接起来实现“**并且**”的效果时，可以使用default方法`and`。其JDK源码为：

```
default Predicate<T> and(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) && other.test(t);
}
```

如果要判断一个字符串既要包含大写“H”，又要包含大写“W”，那么：

```
import java.util.function.Predicate;

public class Demo16PredicateAnd {
    private static void method(Predicate<String> one, Predicate<String> two) {
        boolean isValid = one.and(two).test("Helloworld");
        System.out.println("字符串符合要求吗：" + isValid);
    }

    public static void main(String[] args) {
        method(s -> s.contains("H"), s -> s.contains("W"));
    }
}
```

默认方法：or

与`and`的“与”类似，默认方法`or`实现逻辑关系中的“**或**”。JDK源码为：

```
default Predicate<T> or(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
}
```

如果希望实现逻辑“字符串包含大写H或者包含大写W”，那么代码只需要将“and”修改为“or”名称即可，其他都不变：

```
import java.util.function.Predicate;

public class Demo16PredicateAnd {
    private static void method(Predicate<String> one, Predicate<String> two) {
        boolean isValid = one.or(two).test("Helloworld");
        System.out.println("字符串符合要求吗：" + isValid);
    }

    public static void main(String[] args) {
        method(s -> s.contains("H"), s -> s.contains("W"));
    }
}
```

默认方法：negate

“与”、“或”已经了解了，剩下的“非”（取反）也会简单。默认方法`negate`的JDK源代码为：

```
default Predicate<T> negate() {
    return (t) -> !test(t);
}
```

从实现中很容易看出，它是执行了test方法之后，对结果boolean值进行“!”取反而已。一定要在`test`方法调用之前调用`negate`方法，正如`and`和`or`方法一样：

```
import java.util.function.Predicate;

public class Demo17PredicateNegate {
    private static void method(Predicate<String> predicate) {
        boolean veryLong = predicate.negate().test("HelloWorld");
        System.out.println("字符串很长吗：" + veryLong);
    }

    public static void main(String[] args) {
        method(s -> s.length() < 5);
    }
}
```

1.2 练习：集合信息筛选

题目

数组当中有多条“姓名+性别”的信息如下，请通过`Predicate`接口的拼装将符合要求的字符串筛选到集合`ArrayList`中，需要同时满足两个条件：

1. 必须为女生；
2. 姓名为4个字。

```
public class DemoPredicate {
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };
    }
}
```

解答

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class DemoPredicate {
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };
        List<String> list = filter(array,
                                   s -> "女".equals(s.split(",")[1]),
                                   s -> s.split(",")[0].length() == 3);
        System.out.println(list);
    }

    private static List<String> filter(String[] array, Predicate<String> one,
                                       Predicate<String> two) {
        List<String> list = new ArrayList<>();
        for (String info : array) {
            if (one.and(two).test(info)) {
                list.add(info);
            }
        }
        return list;
    }
}
```

1.3 Function接口

`java.util.function.Function<T,R>`接口用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件。有进有出，所以称为“函数Function”。

抽象方法：apply

`Function`接口中最主要的抽象方法为：`R apply(T t)`，根据类型T的参数获取类型R的结果。使用的场景例如：将`String`类型转换为`Integer`类型。

```
import java.util.function.Function;

public class Demo11FunctionApply {
    private static void method(Function<String, Integer> function) {
        int num = function.apply("10");
        System.out.println(num + 20);
    }

    public static void main(String[] args) {
        method(s -> Integer.parseInt(s));
        method(Integer::parseInt);
    }
}
```

当然，最好是通过方法引用的写法。

默认方法：andThen

`Function`接口中有一个默认的`andThen`方法，用来进行组合操作。JDK源代码如：

```
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T t) -> after.apply(apply(t));
}
```

该方法同样用于“先做什么，再做什么”的场景，和`Consumer`中的`andThen`差不多：

```
import java.util.function.Function;

public class Demo12FunctionAndThen {
    private static void method(Function<String, Integer> one, Function<Integer, Integer> two) {
        int num = one.andThen(two).apply("10");
        System.out.println(num + 20);
    }

    public static void main(String[] args) {
        method(Integer::parseInt, i -> i *= 10);
    }
}
```

第一个操作是将字符串解析成为int数字，第二个操作是乘以10。两个操作通过`andThen`按照前后顺序组合到了一起。

> 请注意，Function的前置条件泛型和后置条件泛型可以相同。

1.4 练习：自定义函数模型拼接

题目

请使用`Function`进行函数模型的拼接，按照顺序需要执行的多个函数操作为：

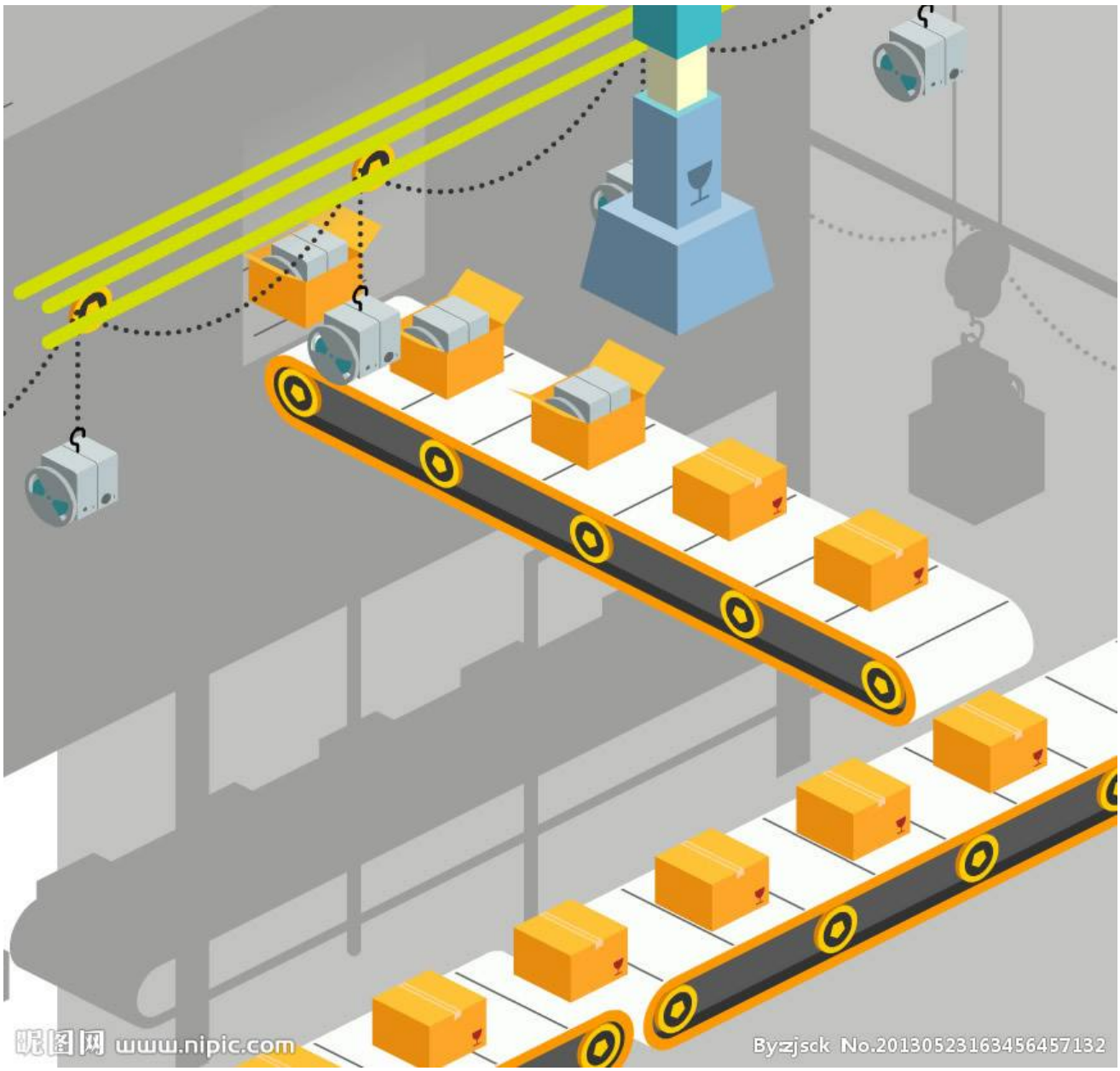
1. 将字符串截取数字年龄部分，得到字符串；
2. 将上一步的字符串转换成为int类型的数字；
3. 将上一步的int数字累加100，得到结果int数字。

解答

```
import java.util.function.Function;

public class DemoFunction {
    public static void main(String[] args) {
        String str = "赵丽颖,20";
        int age = getAgeNum(str, s -> s.split(",")[1],
                            Integer::parseInt,
                            n -> n += 100);
        System.out.println(age);
    }

    private static int getAgeNum(String str, Function<String, String> one,
                                 Function<String, Integer> two,
                                 Function<Integer, Integer> three) {
        return one.andThen(two).andThen(three).apply(str);
    }
}
```

第二章 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中，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既有的弊端。



2.1 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。而当我们需要对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），这点此前已经结合内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，**而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。这是理所当然的么？**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。如果希望再次遍历，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

下面来看一下借助Java 8的Stream API，什么才叫优雅：

```
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);
    }
}
```

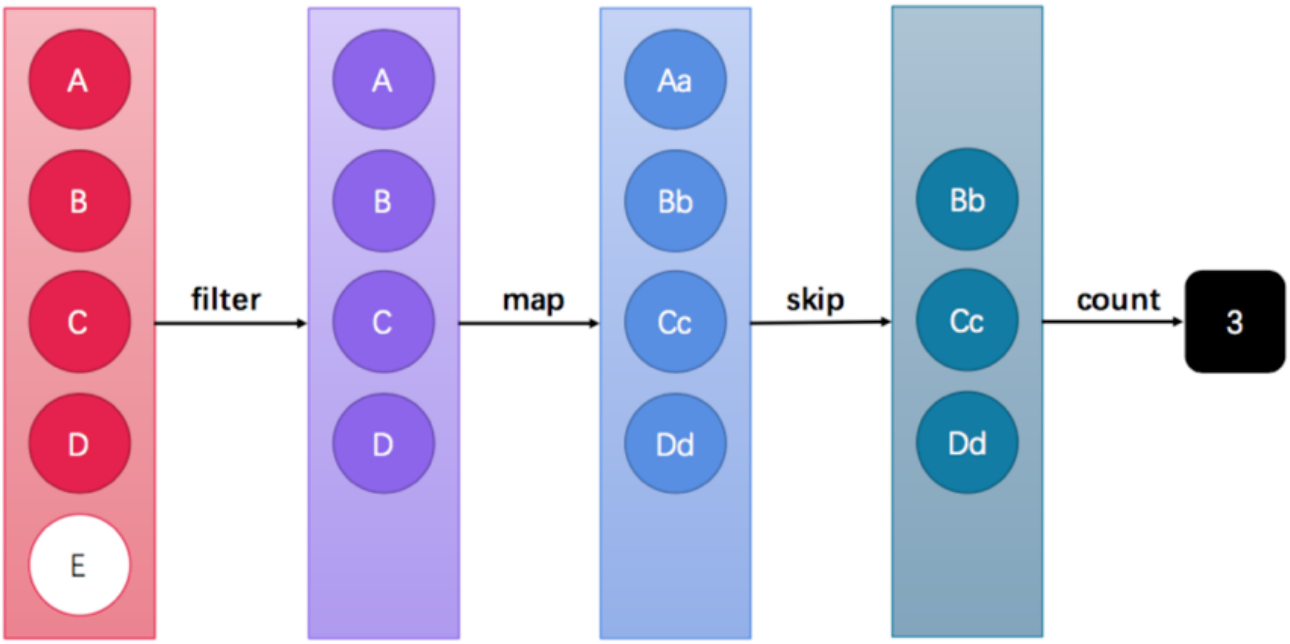
直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：**获取流、过滤姓张、过滤长度为3、逐一打印。**代码中并没有体现使用线性循环或是其他任何算法进行遍历，我们真正要做的事情内容被更好地体现在代码中。

2.2 流式思想概述

注意：请暂时忘记对传统IO流的固有印象！

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个“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，而方案就是一种“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个流模型。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。只有当终结方法`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，其本身并不存储任何元素（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（fluent 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， 这叫做外部迭代。 Stream提供了内部迭代的方式， 通过访问者模式(Visitor)实现。

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。

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`stream`用来获取流，所以其所有实现类均可获取流。

```
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，所以获取对应的流需要分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();
    }
}
```

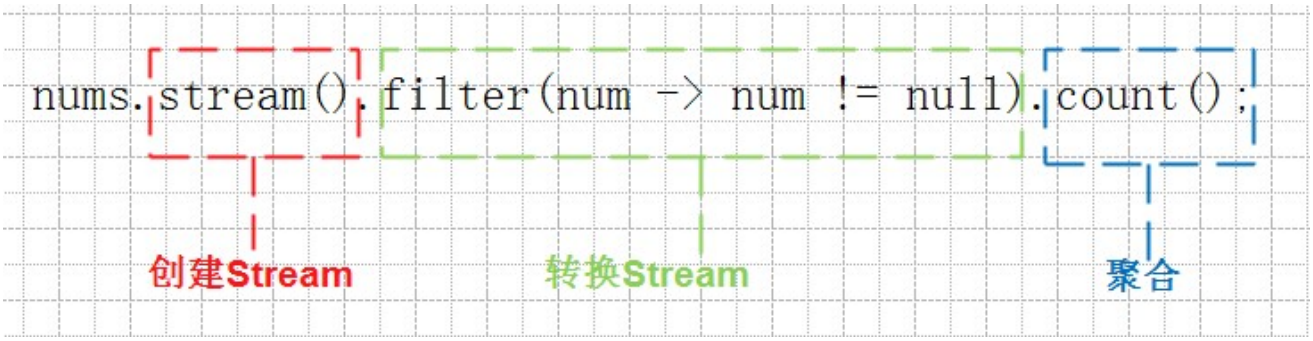
根据数组获取流

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`接口中提供了静态方法`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：

- **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`那样的链式调用。本小节中，终结方法包括`count`和`forEach`方法。
- **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。（除了终结方法外，其余方法均为延迟方法。）

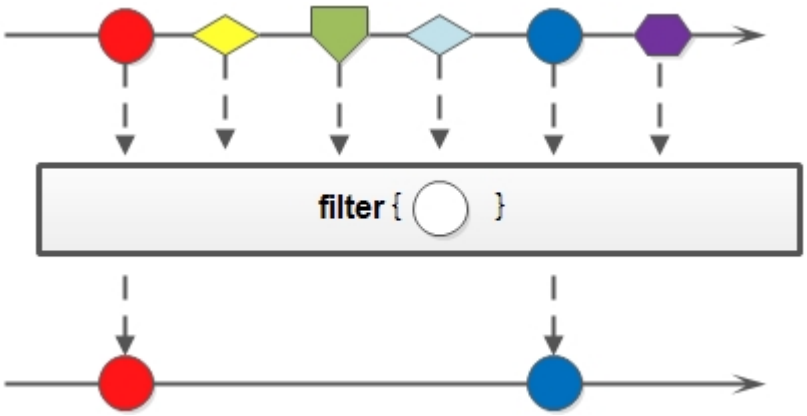
> 备注：本小节之外的更多方法，请自行参考API文档。

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。



复习Predicate接口

此前我们已经学习过`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流的`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。

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：

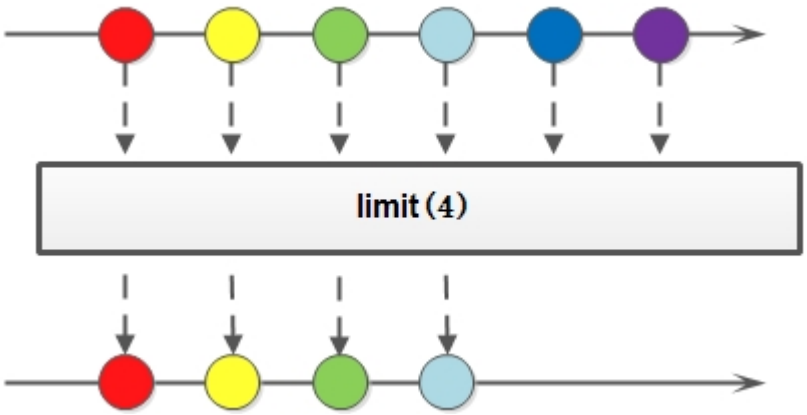
```
import java.util.stream.Stream;

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：



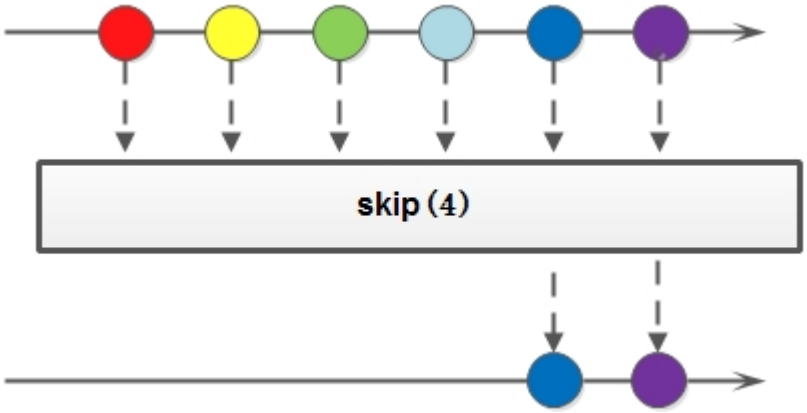

```
import java.util.stream.Stream;

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：



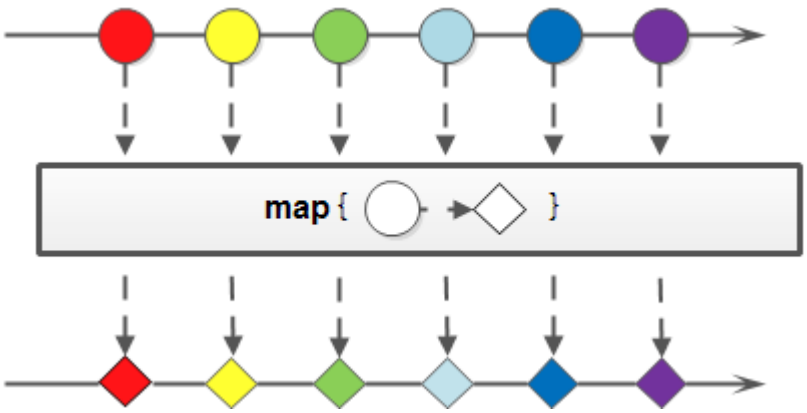

```
import java.util.stream.Stream;

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
    }
}
```

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。



复习Function接口

此前我们已经学习过`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(Integer::parseInt);
    }
}
```

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）。

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);
    }
}
```

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，该方法**并不保证元素的逐一消费动作在流中是被有序执行的。**

```
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。例如：

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

在这里，方法引用`System.out::println`就是一个`Consumer`函数式接口的示例。

2.5 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循环）**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；
2. 第一个队伍筛选之后只要前3个人；
3. 第二个队伍只要姓张的成员姓名；
4. 第二个队伍筛选之后不要前2个人；
5. 将两个队伍合并为一个队伍；
6. 根据姓名创建`Person`对象；
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'}
```

2.6 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'}
```

第三章 扩展知识点

3.1 并发流

当需要对存在于集合或数组中的若干元素进行并发操作时，简直就是噩梦！我们需要仔细考虑多线程环境下的原子性、竞争甚至锁问题，即便是`java.util.concurrent.ConcurrentMap<K, V>`接口也必须谨慎地正确使用。

而对于Stream流来说，这很简单。

转换为并发流

`Stream`的父接口`java.util.stream.BaseStream`中定义了一个`parallel`方法：

```
S parallel();
```

只需要在流上调用一下无参数的`parallel`方法，那么当前流即可变身成为支持并发操作的流，返回值仍然为`Stream`类型。例如：

```
import java.util.stream.Stream;

public class Demo13StreamParallel {
    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50).parallel();
    }
}
```

直接获取并发流

在通过集合获取流时，也可以直接调用`parallelStream`方法来直接获取支持并发操作的流。方法定义为：

```
default Stream<E> parallelStream() {...}
```

应用代码为：

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.stream.Stream;

public class Demo13StreamParallel {
    public static void main(String[] args) {
        Collection<String> coll = new ArrayList<>();
        Stream<String> stream = coll.parallelStream();
    }
}
```

使用并发流

多次执行下面这段代码，结果的顺序在很大概率上是不一定的：

```
import java.util.stream.Stream;

public class Demo13StreamParallel {
    public static void main(String[] args) {
        Stream.of(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
              .parallel()
              .forEach(System.out::println);
    }
}
```

3.2 收集Stream结果

对流操作完成之后，如果需要将其结果进行收集，例如获取对应的集合、数组等，如何操作？

收集到集合中

Stream流提供`collect`方法，其参数需要一个`java.util.stream.Collector<T,A, R>`接口对象来指定收集到哪种集合中。幸运的是，`java.util.stream.Collectors`类提供一些方法，可以作为`Collector`接口的实例：

- `public static <T> Collector<T, ?, List<T>> toList()`：转换为`List`集合。
- `public static <T> Collector<T, ?, Set<T>> toSet()`：转换为`Set`集合。

下面是这两个方法的基本使用代码：

```
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Demo15StreamCollect {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("10", "20", "30", "40", "50");
        List<String> list = stream.collect(Collectors.toList());
        Set<String> set = stream.collect(Collectors.toSet());
    }
}
```

收集到数组中

Stream提供`toArray`方法来将结果放到一个数组中，由于泛型擦除的原因，返回值类型是Object[]的：

```
Object[] toArray();
```

其使用场景如：

```
import java.util.stream.Stream;

public class Demo16StreamArray {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("10", "20", "30", "40", "50");
        Object[] objArray = stream.toArray();
    }
}
```

扩展：解决泛型数组问题

有了Lambda和方法引用之后，可以使用`toArray`方法的另一种重载形式传递一个`IntFunction<A[]>`的函数，继而从外面指定泛型参数。方法签名：

```
<A> A[] toArray(IntFunction<A[]> generator);
```

有了它，上例代码中不再局限于`Object[]`结果，而可以得到`String[]`结果：

```
import java.util.stream.Stream;

public class Demo17StreamArray {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("10", "20", "30", "40", "50");
        String[] strArray = stream.toArray(String[]::new);
    }
}
```

既然数组也是有构造器的，那么传递一个数组的构造器引用即可。

> 备注：Java仍然没有泛型数组，原因同样是泛型擦除。

2.10 练习：将数组元素加到集合中

题目

请通过Stream流的方式，将下面数组当中的元素添加（收集）到`List`集合当中：

```
public class DemoCollect {
    public static void main(String[] args) {
        int[] array = { 10, 20, 30, 40, 50 };
    }
}
```

解答

首先需要将数组转换成为流，然后再通过`collect`方法收集到`List`集合中：

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class DemoCollect {
    public static void main(String[] args) {
        String[] array = { "Java", "Groovy", "Scala", "Kotlin" };
        List<String> list = Stream.of(array).collect(Collectors.toList());
    }
}
```