# JAVA8函数式编程专题

本篇文章基于 B站三更草堂 整理和总结。

## 一、函数式编程思想

面向对象思想需要关注用什么对象完成什么事情，而函数式编程思想就类似于我们数学中的函数，它主要关注对数据进行了什么操作，函数是一等公民。

- 代码简洁，开发迅速
- 大数量下处理集合效率高，易于“并发编程”

其实主要关注它给我们带来的代码简化能力，比如下面的一个代码实现了查询未成年作家评分在70分以上的书籍，并且去重，代码如下：

```
//查询未成年作家的评分在70以上的书籍
List<Book> bookList = new ArrayList<>();
Set<Book> uniqueBookValues = new HashSet<>();
Set<Author> uniqueAuthorValues = new HashSet<>();
for (Author author : authors) {
    if (uniqueAuthorValues.add(author)) {
        if (author.getAge() < 18) {
            List<Book> books = author.getBooks();
            for (Book book : books) {
                if (book.getScore() > 70) {
                    if (uniqueBookValues.add(book)) {
                        bookList.add(book);
                    }
                }
            }
        }
    }
}
System.out.println(bookList);
```

而用Stream流来实现，代码如下：

```
List<Book> collect = authors.stream()//第一步：将作家authors集合转为stream流
    .distinct()//第二步：针对作家authors集合去重
    .filter(author -> author.getAge() < 18)//第三步：筛选未成年作家集合
    .flatMap(author -> author.getBooks().stream())//第四步：获取每个未成年作家对应的作品集合的stream流
    .filter(book -> book.getScore() > 70)//第五步：筛选作品中评分大于70分的作品集合
    .distinct()//第六步：对作品集合再次去重
    .collect(Collectors.toList());//第七步：终结流操作，以List集合格式返回作品集合流
System.out.println(collect);
```

读者看到这两种写法会选择谁？如果没有学习函数式编程，那么可能心中还在犹豫：第一种方式虽然很繁琐，但是我可以写出来；第二种方式看起来简洁很多，但是我有点看不懂、写不出来，这里的写法其实这就是JDK8中的一个语法糖：Lambda表达式，结合专门处理集合或数组的stream流来完成功能，Lambda表达式是JAVA中函数式编程思想的一个重要体现，不关注是什么对象，更关注对数据进行了说明操作。下面我们先来学习Lambda表达式。

# 二、Lambda表达式

Lambda是JDK8中的一个语法糖，他可以对某些匿名内部类的写法进行简化，注意，不是所有匿名内部类，他是函数式编程思想的一个重要体现，让我们不用关注什么式对象，而是更加关注我们对数据进行了什么操作。

基本格式：

> (参数列表)->{代码}

### 例1：我们在创建线程并启动时可以使用匿名内部类写法，这里可以使用Lambda进行简化。

我们以前创建一个线程的写法可以如下：

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("start to do something...");
    }
}).start();
```

我们可以使用Lambda格式对其进行修改，如下：

```
new Thread(()->{
    System.out.println("start to do something...");
}).start();
```

可以看到，省去了不少格式化的代码，使得整体可读性得到了提升。

其实，如果这个匿名内部类是一个接口、并且这个接口下面只有一个抽象方法的接口需要实现时，就可以通过Lambda来简化，比如这里的Runnable接口：

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface
<code>Runnable</code> is used
     * to create a thread, starting the thread causes the
object's
     * <code>run</code> method to be called in that separately
executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is
that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

大家可能会注意到@FunctionalInterface这个注解，中文名叫函数式接口，是JAVA8中对一类特殊类型接口的称呼，这类接口只定义了唯一的抽象方法的接口，这类接口就可以使用Lambda来简化实现。这个注解其实是一种契约，当带上这个注解时，就需要符合约定，比如只能标记在"有且仅有一个抽象方法"的接口上，否则编译会不通过。JAVA中有不少这样的函数式接口，下面我们结合例子实际看一看。

### 例2：完成两个整数相加的功能

首先自定义一个叫做calculateNum的函数，里面传参IntBinaryOperator，最后return一个operator.applyAsInt(a,b)的结果。

```
public static int calculateNum(IntBinaryOperator operator){
    int a = 10;
    int b = 20;
    return operator.applyAsInt(a,b);
}
```

首先抛开函数功能，我们来看下IntBinaryOperator，点进源码可以看到定义：

```
/**
 * Represents an operation upon two int-valued operands and
producing an int-valued result. This is the primitive type
specialization of BinaryOperator for int.
 */
@FunctionalInterface
public interface IntBinaryOperator {

    /**
     * Applies this operator to the given operands.
     *
     * @param left the first operand
     * @param right the second operand
     * @return the operator result
     */
    int applyAsInt(int left, int right);
}
```

是一个标准的函数式接口，里面有一个抽象方法叫做applyAsInt等待我们来重写，根据注释说明，这个函数式接口的含义仅仅是对两个整数进行操作并生成一个新的整数结果，那么我们就先使用匿名内部类来重写实现两个整数相加的功能。

```
public static void main(String[] args) {
    int result = calculateNum(new IntBinaryOperator() {
        @Override
        public int applyAsInt(int left, int right) {
            return left + right;
        }
    });
    System.out.println("the result = " + result);
}
```

输出结果为30，符合预期，下面我们将其改为Lambda表达式：

```
public static void main(String[] args) {
    int result = calculateNum((int left, int right)->{return left
+ right;});
    System.out.println("the result = " + result);
}
```

可以看到，符合(参数列表)->{代码}格式即可，左边是匿名内部类的入参，右边是参数的操作。

是不是有点感觉了？在IDEA中多敲几遍回味回味。当然了，熟练以后可以通过IDEA的快捷键快速由匿名内部类改写为Lambda表达式，也可以通过快捷键快速由Lambda表达式恢复为匿名内部类写法。

**例3：现有方法定义如下，其中IntPredicate是一个接口，先使用匿名内部类的写法调用该方法，再改写为Lambda表达式写法调用，实现偶数打印、奇数不打印功能。**

```
public static void printNum(IntPredicate predicate){
    int[] arr = {1,2,3,4,5,6,7,8,9,10};
    for (int i : arr){
        if (predicate.test(i)){
            System.out.println(i);
        }
    }
}
```

我们点进IntPredicate接口，如下：

```
@FunctionalInterface
public interface IntPredicate {

    boolean test(int value);

    default IntPredicate and(IntPredicate other) {
        Objects.requireNonNull(other);
        return (value) -> test(value) && other.test(value);
    }

    default IntPredicate negate() {
        return (value) -> !test(value);
    }
```

```
    default IntPredicate or(IntPredicate other) {
        Objects.requireNonNull(other);
        return (value) -> test(value) || other.test(value);
    }
}
```

可以看到IntPredicate是一个函数式接口，里面只有一个test抽象方法，其他方法是默认方法，不算抽象方法，所以与上面定义并不冲突。调用printNum函数需要传入IntPredicate接口对象，我们需要在传入的同时重写里面的test方法实现是否是偶数的判断条件，当printNum函数中遍历每个元素时，都会将元素传到匿名内部类中执行拿到结果：

```
printNum(new IntPredicate() {
    @Override
    public boolean test(int value) {
        //如果是偶数，则符合判断条件，打印出来
        return value%2 == 0;
    }
});
```

通过Lambda表达式改写，按照上面规则我们应该可以很容易写出来：

```
printNum((int value) -> {return value%2 == 0;});
```

**例4：现有方法如下，其中IntConsumer是一个函数式接口，来尝试使用匿名内部类写法和Lambda表达式写法分别调用该方法。**

```
public static void foreachArr(IntConsumer consumer){
    int[] arr = {1,2,3,4,5,6,7,8,9,10};
    for (int i : arr){
        consumer.accept(i);
    }
}
```

我们点进IntConsumer函数：

```
@FunctionalInterface
public interface IntConsumer {

    void accept(int value);

    ...
}
```

首先accept方法需要一个int类型的入参，没有返回值，基本可以猜出，其实主要是对int入参做一些判断，但是我们不能做转换返回，那么我们就可以利用它实现简单的偶数打印、奇数不打印的效果，如下使用匿名内部类写法：

```
foreachArr(new IntConsumer() {
    @Override
    public void accept(int value) {
        if (value%2 == 0){
            System.out.println(value);
        }
    }
});
```

在foreachArr函数中，for循环调用consumer.accept(i)时，就会将i传入到匿名内部类中做处理。

通过Lambda表达式改写，按照上面规则我们应该可以很容易写出来：

```
foreachArr((int value) -> {
    if (value%2 == 1){
        System.out.println(value);
    }
});
```

**例5：现有方法如下，其中Function是一个函数式接口，来尝试使用匿名内部类写法和Lambda表达式写法分别调用该方法。**

```
public static <R> R typeConver(Function<String,R> function){
    String str = "1234";
    return function.apply(str);
}
```

我们点进Function函数，根据注释，说这个函数的作用是接收一个参数，返回一个结果，十分地简单。

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    ...
}
```

接收参数类型和返回参数类型都可以自定义，接收的范型叫T，返回的范型叫R，也就是说，使用Function函数，你可以任意传入一个类型比如是String字符串，返回一个整数类型或者字符串类型，就看你的定义。

如下，传入一个String类型“1234”，返回一个Integer类型1234，使用匿名内部类写法：

```
Integer result = typeConver(new Function<String, Integer>() {
    @Override
    public Integer apply(String s) {
        return Integer.valueOf(s);
    }
});

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

通过Lambda表达式改写，按照上面规则我们应该可以很容易写出来：

```
Integer result = typeConver((String s) -> {
        return Integer.valueOf(s);
});
System.out.println(result);
```

其实Lambda表达式可以做进一步的省略。

- 参数类型可以省略，因为函数式接口就一个抽象方法，参数类型本来就是可以由代码自动识别出来的，因此可以省略；
- 方法体只有一句代码时，大括号、return和唯一一句代码的分号可以省略；
- 方法只有一个参数时小括号可以省略；

对于例5，我们可以看到，方法体只有一行，那么就可以做简化，简化完后的代码为：

```
Integer result = typeConver(s -> Integer.valueOf(s));
System.out.println(result);
```

如上代码所示，省略了s的参数类型String，也省略了return、大括号、方法最后的分号。按照这个规则，可以轻松写出简化写法。

如果你使用IDEA这样的工具，使用快捷键可以自动完成Lambda表达式的转换以及简化，所以其实重点是理解Lambda表达式世界的规则、能轻松看懂Lambda表达式代码，剩下的就是让IDEA为我们提高生产效率。

通过上面几个简单的例子可以有这么一个感受，其实Lambda表达式就是对匿名内部类写法的优化，作为初学者，我们应该先用匿名内部类写法写出来，再用Lambda表达式优化，这样就会自然而然清晰地理解Lambda表达式的优势，从而主动拥抱Lambda表达式。

# 三、Stream流

JAVA8的Stream流使用的是函数式编程模式，如同它的名字一样，它可以被用来**对集合或数组进行链状流式处理，可以更方便地让我们对集合或数组操作**。

整体来说，对Stream流的操作分为三类：创建流、流的中间操作、流的终结操作。

流的中间操作有：filter、map、distinct、sorted、limit、skip、flatMap

流的终结操作有：forEach、count、max&min、collect、查找和匹配（anyMatch、allMatch、noneMatch、findAny、findFirst）、reduce归并

## 提前准备

```
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.16</version>
    </dependency>
</dependencies>
```

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode//用于后期的去重使用
public class Author {
    //id
    private Long id;
    //姓名
    private String name;
    //年龄
```

```
    private Integer age;
    //简介
    private String intro;
    //作品
    private List<Book> books;
}
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode//用于后期的去重使用
public class Book {
    //id
    private Long id;
    //书名
    private String name;

    //分类
    private String category;

    //评分
    private Integer score;

    //简介
    private String intro;

}
```

```
private static List<Author> getAuthors() {
    //数据初始化
    Author author = new Author(1L,"蒙多",33,"一个从菜刀中明悟哲理的祖
安人",null);
    Author author2 = new Author(2L,"亚拉索",15,"狂风也追逐不上他的思考
速度",null);
    Author author3 = new Author(3L,"易",14,"是这个世界在限制他的思
维",null);
    Author author4 = new Author(3L,"易",14,"是这个世界在限制他的思
维",null);

    //书籍列表
    List<Book> books1 = new ArrayList<>();
    List<Book> books2 = new ArrayList<>();
    List<Book> books3 = new ArrayList<>();
```

```
    books1.add(new Book(1L,"刀的两侧是光明与黑暗","哲学,爱情",88,"用一把刀划分了爱恨"));
    books1.add(new Book(2L,"一个人不能死在同一把刀下","个人成长,爱情",99,"讲述如何从失败中明悟真理"));

    books2.add(new Book(3L,"那风吹不到的地方","哲学",85,"带你用思维去领略世界的尽头"));
    books2.add(new Book(3L,"那风吹不到的地方","哲学",85,"带你用思维去领略世界的尽头"));
    books2.add(new Book(4L,"吹或不吹","爱情,个人传记",56,"一个哲学家的恋爱观注定很难把他所在的时代理解"));

    books3.add(new Book(5L,"你的剑就是我的剑","爱情",56,"无法想象一个武者能对他的伴侣这么的宽容"));
    books3.add(new Book(6L,"风与剑","个人传记",100,"两个哲学家灵魂和肉体的碰撞会激起怎么样的火花呢？"));
    books3.add(new Book(6L,"风与剑","个人传记",100,"两个哲学家灵魂和肉体的碰撞会激起怎么样的火花呢？"));

    author.setBooks(books1);
    author2.setBooks(books2);
    author3.setBooks(books3);
    author4.setBooks(books3);

    List<Author> authorList = new ArrayList<>(Arrays.asList(author,author2,author3,author4));
    return authorList;
}
```

## 案例快速入门

需求：打印所有年龄小于18的作家的名字，并且要注意去重。

传统实现方式：

```
//打印所有年龄小于18的作家的名字，并且要注意去重
List<Author> authors = getAuthors();
Set<Author> uniqueAuthorValues = new HashSet<>();
for (Author author : authors){
    if (!uniqueAuthorValues.contains(author)) {
        if (author.getAge() < 18) {
            uniqueAuthorValues.add(author);
            System.out.println(author.getName());
        }
    }
}
```

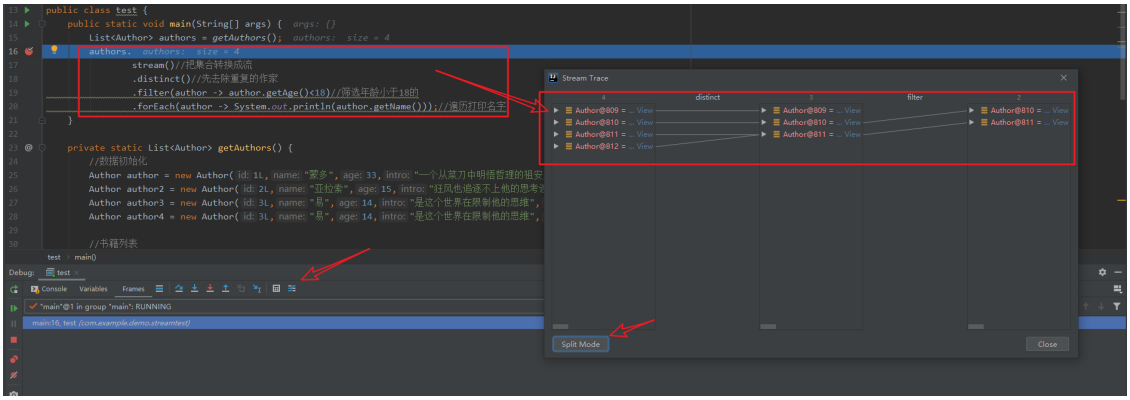
如上，我们需要使用Set去重特性的集合来实现去重逻辑，然后for循环遍历加条件判断，代码十分繁琐，如果条件再复杂一点，可想而知，代码可读性将急剧下降。

使用Stream流来实现这个需求：

```
//打印所有年龄小于18的作家的名字，并且要注意去重
List<Author> authors = getAuthors();
authors.
    stream()//把集合转换成流
    .distinct()//先去除重复的作家
    .filter(author -> author.getAge()<18)//筛选年龄小于18的
    .forEach(author -> System.out.println(author.getName()));//遍历打印名字
```

可以看到，用Stream流来处理集合类型的元素十分简洁，使用这种方式操作集合，其实也大大降低了bug的发生，因为for+if的堆叠，一不小心就会写错逻辑。

但是有的同学会说这种写法调试并不友好，其实IDEA已经十分强大，可以帮助我们通过可视化页面查看流的处理中间流程。（不过有些写法确实不好调试）




经过如上体验，我们发现一旦掌握了Stream流的写法，确实会大大提高生产效率，还能帮助我们降低代码编写的错误率，下面我们就来好好学习下Stream流中的各种常用操作吧。

## Stream流操作-创建流

既然是Stream流操作，第一步自然是创建流，再对流做后续的处理，我们说过，Stream流操作的对象是集合或数组，那么问题变成：集合或数组是如何创建流的呢?

如果是单列集合，比如典型的List集合，存放的是单个元素，创建流的方式是：

> 集合对象.stream()

例如：

```
List<Author> authors = getAuthors();
Stream<Author> stream = authors.stream();
```

如果是数组，创建流的方式是：

> Arrays.stream(数组)或者使用Stream.of来创建

```
Integer[] arr = {1,2,3,4,5};
Stream<Integer> stream = Arrays.stream(arr);
Stream<Integer> stream2 = Stream.of(arr);
```

对于双列集合典型如MAP，转换成单列集合后再创建，比如MAP中的Entry对象，是封装了键和值的对象，创建流的方式是：

```
Map<String,Integer> map = new HashMap<>();
map.put("蜡笔小新",19);
map.put("黑子",17);
map.put("日向翔阳",16);

Stream<Map.Entry<String, Integer>> stream =
map.entrySet().stream();
```

如上所示代码，map.entrySet()返回的对象是Set<Map.Entry<K, V>>，而Set单列集合本身就有.stream()方法将其转为流。

# Stream流的中间操作和终结操作

## filter、distinct、forEach的使用

- 【中间操作】filter作用：对流中的元素进行条件过滤，符合过滤条件的元素才能留在流中。
- 【中间操作】distinct作用：去流中的元素进行去重，留下去重后的元素
- 【终结操作】forEach作用：遍历留在流中的元素，不会对元素做任何改动

```
//打印所有年龄小于18的作家的名字
List<Author> authors = getAuthors();
authors.
    stream()//把集合转换成流
    .distinct()//先去除重复的作家
    .filter(author -> author.getAge()<18)//筛选年龄小于18的
    .forEach(author -> System.out.println(author.getName()));//遍
历打印名字
```

其中，使用distinct的时候，我们需要让Lambda知道对象是否重复的判断依据，其实就是按照对象Object的equals方法进行判断的，如果是我们自定义的对象，一定要注意重写equals方法来实现正确的判断！

## sorted、limit的使用

- 【中间操作】sorted作用：对流中元素进行排序
- 【中间操作】limit作用：对返回的元素个数进行限制

需求：打印作家的名字，要求：元素不重复，且年龄降序排列，展示前两个作家即可

我们可以使用sorted来排序，使用limit来限制返回元素个数：

//打印作家的名字，要求：元素不重复，且年龄降序排列，展示前两个作家即可：

```
List<Author> authors = getAuthors();
authors.stream()//把集合转换成流
        .distinct()//先去除重复的作家，剩下
        .sorted()
        .limit(2)
        .forEach(author ->
System.out.println(author.getName()));//遍历打印名字
```

运行会报错如下：

```
Exception in thread "main" java.lang.ClassCastException: class
com.example.streamstudy.Author cannot be cast to class
java.lang.Comparable (com.example.streamstudy.Author is in
unnamed module of loader 'app'; java.lang.Comparable is in module
java.base of loader 'bootstrap')
```

其实，如果调用空参的sorted()方法，需要流中的元素是实现了Comparable接口，这里需要对Author对象增加实现Comparable接口的重写方法，如下：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode//用于后期的去重使用
public class Author implements Comparable<Author>{
    //id
    private Long id;
    //姓名
    private String name;
    //年龄
    private Integer age;
    //简介
    private String intro;
    //作品
    private List<Book> books;

    @Override
    public int compareTo(Author o) {
        return o.age - this.age;
    }
}
```

在JAVA中，实现对象比较，有两种方法，一种就是我们自己可掌控的对象，我们可以在对象类中进行如上的改写。但是有的对象是不可改写的，如何做呢？那就需要在sorted函数中增加对象的比较规则参数。

```
//打印作家的名字，要求：元素不重复，且年龄降序排列，展示前两个作家即可
List<Author> authors = getAuthors();
authors.stream()//把集合转换成流
        .distinct()//先去除重复的作家，还剩下3个作家
        .sorted((o1, o2) -> o2.getAge()-o1.getAge())//按照作家年龄降
序排列
        .limit(2)//限制最终返回前两个2作家
        .forEach(author ->
System.out.println(author.getName()));//遍历打印名字
```

## skip的使用

- 【中间操作】skip作用：顾名思义，跳过某些元素，这里是跳过流中前N个元素，返回剩下的元素。

需求：打印作家的名字，要求：元素不重复，且年龄降序排列，展示的作家中，去除年纪最大的一个人，返回剩下的作家

```
List<Author> authors = getAuthors();
authors.stream()
        .distinct()
        .sorted((o1, o2) -> o2.getAge()-o1.getAge())
        .skip(1)
        .forEach(author -> System.out.println(author.getName()));
```

## map、collect的使用

- 【中间操作】map作用：可以对流中元素进行计算或数据类型转换。
- 【终结操作】collect作用：将当前流转换成一个集合

来个示例：

```
List<Author> authors = getAuthors();
    List<Integer> ageList = new ArrayList<>();
    ageList = authors.stream()
            .map(author -> author.getAge())
            .collect(Collectors.toList());
    System.out.println(ageList);
```

如上，我们完成了一个目标：将List对象转为了List对象。

collect作用是将当前流转换成一个集合，那么我们可以完成各种集合之间的互相转换，比如list转set，list转map等。我们以list转map为例，key为作家名字name，value为该作家对应的作品集合：

```
List<Author> authors = getAuthors();
Map<String,List<Book>> map = new HashMap<>();
map = authors.stream()
        .distinct()
        .collect(Collectors.toMap(author ->
author.getName(),author -> author.getBooks()));
System.out.println(map);
```

## flatMap

- 【中间操作】flatMap作用： map只能把一个对象转换成另一个对象来作为流中的元素。而flatMap可以把一个对象转换成多个对象作为流中的元素。

这个相较于前面的几个操作稍微有点复杂，我们通过具体案例来理解。

例一：打印所有书籍的名字。要求对重复的元素进行去重。

分析：我们需要遍历所有的作家，而每个作家又有多个作品，因此我们需要在遍历每个作家时，将其转成多个作品对象，这里就会涉及到将一个作家对象转为多个作品对象，需要使用flatMap。

```
List<Author> authors = getAuthors();
authors.stream()
        .distinct()
        .flatMap(author -> author.getBooks().stream())
        .distinct()
        .forEach(book -> System.out.println(book.getName()));
```

例二：打印现有作品的所有分类。要求对分类进行去重。原始数据中，可能有多个分类，之间通过逗号分割。

```
List<Author> authors = getAuthors();
authors.stream()
        .flatMap(author -> author.getBooks().stream())
        .distinct()
        .flatMap(book ->
Arrays.stream(book.getCategory().split(",")))
        .distinct()
        .forEach(category -> System.out.println(category));
```

## count、min&max的使用

- 【终结操作】count作用：可以用来获取当前流中元素的个数。
- 【终结操作】min作用：用来获取流中元素最大值
- 【终结操作】max作用：用来获取流中元素最小值

例1：获取作家所有书籍的数目，注意删除重复元素。

```
List<Author> authors = getAuthors();
long count = authors.stream()
        .flatMap(author -> author.getBooks().stream())
        .distinct()
        .count();
System.out.println("不重复的作品数量为: " + count);
```

例2：分别获取这些作家的所出书籍的最高分和最低分并打印。

```
List<Author> authors = getAuthors();
Optional<Integer> max = authors.stream()
        .flatMap(author -> author.getBooks().stream())
        .map(book -> book.getScore())
        .max((score1, score2) -> score1 - score2);
System.out.println(max.get());
Optional<Integer> min = authors.stream()
        .flatMap(author -> author.getBooks().stream())
        .map(book -> book.getScore())
        .min((score1, score2) -> score1 - score2);
System.out.println(min.get());
```

这里使用的了后面即将学习的Optional，这里我们只要知道，max或min的返回类型是它，并且可以通过get方法获取结果即可。

## 查找与匹配

- 【终结操作】anyMatch作用：可以用来判断是否有任意符合匹配条件的元素，结果为boolean类型。
- 【终结操作】allMatch作用：可以用来判断是否都符合匹配条件，结果为boolean类型。如果都符合结果为true，否则结果为false。
- 【终结操作】noneMatch作用：可以判断流中的元素是否都不符合匹配条件。如果都不符合结果为true，否则结果为false
- 【终结操作】findAny作用：获取流中的任意一个元素，该方法没有办法保证获取的一定是流中的第一个元素。
- 【终结操作】findFirst作用：获取流中的第一个元素。

例1：判断是否有年龄在18岁及以上的作家，只要有一个就为true，使用anyMatch。

```
boolean flag1 = authors.stream()
        .anyMatch(author -> author.getAge() >= 18);
System.out.println("是否有年龄在18岁及以上的作家:" + flag1);
```

例2：判断是否所有的作家都是成年人，使用allMatch。

```
boolean flag2 = authors.stream()
        .allMatch(author -> author.getAge() >= 18);
System.out.println("判断是否所有的作家都是成年人:" + flag2);
```

例3：判断作家是否都没有超过100岁。

```
boolean flag3 = authors.stream()
        .noneMatch(author -> author.getAge() > 100);
System.out.println("判断作家是否都没有超过100岁:" + flag3);
```

例4：获取任意一个年龄大于18的作家，如果存在就输出他的名字。

```
Optional<Author> optionalAuthor = authors.stream()
                .filter(author -> author.getAge()>18)
                .findAny();
optionalAuthor.ifPresent(author -> System.out.println("获取任意一个
年龄大于18的作家，如果存在就输出他的名字:" + author.getName()));
```

例5：获取一个年龄最小的作家，并输出他的姓名。

```
Optional<Author> first = authors.stream()
        .sorted((o1, o2) -> o1.getAge() - o2.getAge())
        .findFirst();
first.ifPresent(author -> System.out.println("获取一个年龄最小的作
家，并输出他的姓名:" + author.getName()));
```

## reducde的使用

- 【终结操作】reduce作用：对流中的数据按照你指定的计算方式计算出一个结果。

展开来讲，reduce的作用是把stream中的元素给组合起来，我们可以传入一个初始值，它会按照我们的计算方式依次拿流中的元素和初始化值进行计算，计算结果再和后面的元素计算。

reduce两个参数的重载形式内部的计算方式如下：

```
T result = identity;
for (T element : this stream)
  result = accumulator.apply(result, element)
return result;
```

其中identity就是我们可以通过方法参数传入的初始值，accumulator的apply具体进行什么计算也是我们通过方法参数来确定的。

其实可以初步看出，reducde其实就是对for循环计算的一个封装，帮助我们减少代码量。

例：使用reduce求所有作者年龄的和

```
List<Author> authors = getAuthors();
Integer count = authors.stream()
        .distinct()
        .map(author -> author.getAge())//转为List<Integer>类型
        .reduce(0, (result, element) -> (result +
element));//result为结果变量，0是初始值identity，element是集合元素值
System.out.println(count);
```

使用Stream流的几个注意点：

- 惰性求值：如果没有终结操作，中间操作是不会得到执行的。
- 流是一次性的：一旦一个流对象经过一个终结操作后。这个流就不能再被使用。
- 不会影响原数据：我们在流中可以多数据做很多处理。但是正常情况下是不会影响原来集合中的元素的，这往往也是我们期望的。

# 四、Optional

我们在编写代码的时候出现最多的就是空指针异常。所以在很多情况下我们需要做各种非空的判断。

```
Author author = getAuthor();
if(author!=null){
    System.out.println(author.getName());
}
```

尤其是对象中的属性还是一个对象的情况下，这种判断会更多，而过多的判断语句会让我们的代码显得臃肿不堪。

所以在JDK8中引入了Optional,养成使用Optional的习惯后你可以写出更优雅的代码来避免空指针异常，并且在很多函数式编程相关的API中也都用到了Optional，如果不会使用Optional也会对函数式编程的学习造成影响。

### 创建对象

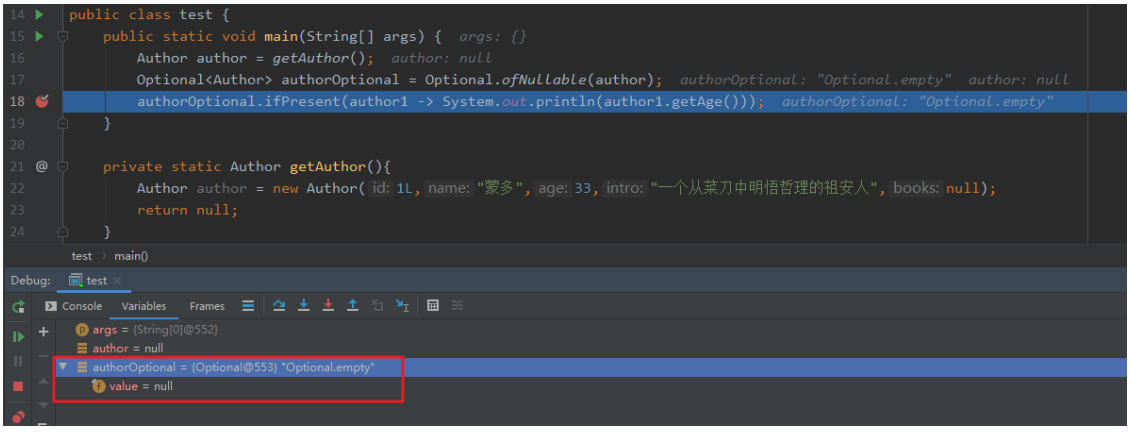
Optional就好像是包装类，可以把我们的具体数据封装Optional对象内部。然后我们去使用Optional中封装好的方法操作封装进去的数据就可以非常优雅的避免空指针异常。

我们一般使用Optional的静态方法ofNullable来把数据封装成一个Optional对象，无论传入的参数是否为null都不会出现问题。

```
public static void main(String[] args) {
    Author author = getAuthor();
    Optional<Author> authorOptional =
Optional.ofNullable(author);
    authorOptional.ifPresent(author1 ->
System.out.println(author1.getAge()));
}

private static Author getAuthor(){
    Author author = new Author(1L,"蒙多",33,"一个从菜刀中明悟哲理的祖
安人",null);
    return null;
}
```

这里getAuthor()返回的是null，运行main并不会报错，authorOptional.ifPresent判断为空后就不会执行后续打印操作。




你可能会觉得还要加一行代码来封装数据比较麻烦。但是如果改造下getAuthor方法，让其的返回值就是封装好的Optional的话，我们在使用时就会方便很多，如下：

```
public static void main(String[] args) {
    Optional<Author> authorOptional = getAuthor();
    authorOptional.ifPresent(author1 ->
System.out.println(author1.getAge()));
}

private static Optional<Author> getAuthor(){
    Author author = new Author(1L,"蒙多",33,"一个从菜刀中明悟哲理的祖
安人",null);
    Optional<Author> authorOptional =
Optional.ofNullable(author);
    return authorOptional;
}
```

而且在实际开发中我们的数据很多是从数据库获取的。Mybatis从3.5版本可以也已经支持Optional了。我们可以直接把dao方法的返回值类型定义成Optional类型，MyBastis会自己把数据封装成Optional对象返回。封装的过程也不需要我们自己操作。

如果你确定一个对象不是空的则可以使用Optional的静态方法of来把数据封装成Optional对象。

```
public static void main(String[] args) {
    Optional<Author> authorOptional = getAuthor();
    authorOptional.ifPresent(author1 ->
System.out.println(author1.getAge()));
}

private static Optional<Author> getAuthor(){
    Author author = new Author(1L,"蒙多",33,"一个从菜刀中明悟哲理的祖
安人",null);
    author = null;
    Optional<Author> authorOptional = Optional.of(author);
    return authorOptional;
}
```

但是一定要注意，如果使用of的时候传入的参数必须不为null，这里故意传递一个null，运行代码报错；

```
Exception in thread "main" java.lang.NullPointerException
  at java.base/java.util.Objects.requireNonNull(Objects.java:208)
  at java.base/java.util.Optional.of(Optional.java:113)
  at
com.example.streamstudy.StreamTest.getAuthor(StreamTest.java:150)
  at com.example.streamstudy.StreamTest.main(StreamTest.java:143)
```

关于ofNullable和of两个静态方法的区别，其实只需要点开源码就可以看出来区别，ofNullable方法源码如下：

```
/**
 * Returns an {@code Optional} describing the given value, if
 * non-{@code null}, otherwise returns an empty {@code Optional}.
 *
 * @param value the possibly-{@code null} value to describe
 * @param <T> the type of the value
 * @return an {@code Optional} with a present value if the
specified value
 *         is non-{@code null}, otherwise an empty {@code
Optional}
 */
@SuppressWarnings("unchecked")
public static <T> Optional<T> ofNullable(T value) {
    return value == null ? (Optional<T>) EMPTY
                         : new Optional<>(value);
}
```

可以看到，首先对value做了非空判断，如果是空，则返回一个EMPTY，这个EMPTY的定义是：

```
private static final Optional<?> EMPTY = new Optional<>(null);
```

因此我们即便传入的是null也不会报错。而of的源码：

```
public static <T> Optional<T> of(T value) {
    return new Optional<>(Objects.requireNonNull(value));
}

public static <T> T requireNonNull(T obj) {
    if (obj == null)
        throw new NullPointerException();
    return obj;
}
```

当传入的value为空时会抛出NullPointerException异常，需要提前做下判断或捕获异常，比较麻烦，失去了本身引入Optional的初衷，因此我们建议使用ofNullable方法。

### 安全消费值

我们获取到一个Optional对象后肯定需要对其中的数据进行使用。这时候我们可以使用其ifPresent方法对来消费其中的值。

这个方法会判断其内封装的数据是否为空，不为空时才会执行具体的消费代码。这样使用起来就更加安全了。

例如,以下写法就优雅的避免了空指针异常。

```
Optional<Author> authorOptional =
Optional.ofNullable(getAuthor());

authorOptional.ifPresent(author ->
System.out.println(author.getName()));
```

### 获取值&安全获取值

获取值：如果我们想获取值自己进行处理可以使用get方法获取，但是不推荐。因为当Optional内部的数据为空的时候会出现异常。例如下面的写法：

```
public static void main(String[] args) {
    Optional<Author> authorOptional = getAuthor();
    Author author = authorOptional.get();
}

private static Optional<Author> getAuthor(){
    Optional<Author> authorOptional = Optional.ofNullable(null);
    return authorOptional;
}
```

抛出异常：

```
Exception in thread "main" java.util.NoSuchElementException: No
value present
  at java.base/java.util.Optional.get(Optional.java:143)
  at com.example.streamstudy.StreamTest.main(StreamTest.java:147)
```

如果我们期望安全的获取值，我们不推荐使用get方法，而是使用Optional提供的以下方法。

- orElseGet：获取数据并且设置数据为空时的默认值，如果数据不为空就能获取到该数据。如果为空则根据你传入的参数来创建对象作为默认值返回。
- orElseThrow：获取数据，如果数据不为空就能获取到该数据，如果为空则根据你传入的参数来创建异常抛出。

orElseGet示例：

```
Optional<Author> authorOptional = getAuthor();
Author author = authorOptional.orElse(new Author());
```

也可以使用orElseThrow抛出自定义异常，用于捕获和处理：

```
Optional<Author> authorOptional = getAuthor();
try {
    Author author =
authorOptional.orElseThrow((Supplier<Throwable>) () -> new
RuntimeException("author为空"));
    System.out.println(author.getName());
}catch (Throwable throwable) {
    throwable.printStackTrace();
}
```

### 过滤

我们可以使用filter方法对数据进行过滤。如果原本是有数据的，但是不符合判断，也会变成一个无数据的Optional对象。

示例如下：

```
Optional<Author> authorOptional = getAuthor();
authorOptional.filter(author ->
author.getAge()>20).ifPresent(author ->
System.out.println(author.getName()));
```

### 数据转换

Optional还提供了map可以让我们的对数据进行转换，并且转换得到的数据也还是被Optional包装好的，保证了我们的使用安全。

例如我们想获取作家的书籍集合。

```
private static void testMap() {
    Optional<Author> authorOptional = getAuthor();
    Optional<List<Book>> optionalBooks =
authorOptional.map(author -> author.getBooks());
    optionalBooks.ifPresent(books -> System.out.println(books));
}
```

# 五、函数式接口

我们上面已经学习过了函数式接口，有如下：

- Consumer消费接口
- Function计算转换接口
- Predicate判断接口
- ...

我们尤其关注下Predicate判断接口，它还有一些默认方法我们可以使用：

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

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

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

我们来解释下：

- and：我们在使用Predicate接口时候可能需要进行判断条件的拼接，而and方法相当于是使用&&来拼接两个判断条件。
- or：我们在使用Predicate接口时候可能需要进行判断条件的拼接，而or方法相当于是使用||来拼接两个判断条件。
- negate：negate方法相当于是在判断添加前面加了个！表示取反。

例1：打印作家中年龄大于17并且姓名的长度大于1的作家。

```
List<Author> authors = getAuthors();
Stream<Author> authorStream = authors.stream();
authorStream.filter(new Predicate<Author>() {
    @Override
    public boolean test(Author author) {
        return author.getAge()>17;
    }
}.and(new Predicate<Author>() {
    @Override
    public boolean test(Author author) {
        return author.getName().length()>1;
    }
})).forEach(author -> System.out.println(author));
```

通过IDEA快捷方式将其转换为Lambda表达式写法：

```
List<Author> authors = getAuthors();
        Stream<Author> authorStream = authors.stream();
        authorStream.filter(((Predicate<Author>) author ->
author.getAge() > 17).and(author ->
author.getName().length()>1)).forEach(author ->
System.out.println(author));
```

当然了，还可以使用&&更加简化：

```
List<Author> authors = getAuthors();
Stream<Author> authorStream = authors.stream();
authorStream.filter(author -> author.getAge() > 17 &&
author.getName().length()>1).forEach(author ->
System.out.println(author));
```

不过默认函数还是建议使用第一种匿名内部类的格式来写，比较清晰。

例2：打印作家中年龄大于17或者姓名的长度小于2的作家。

```
//        打印作家中年龄大于17或者姓名的长度小于2的作家。
List<Author> authors = getAuthors();
authors.stream()
        .filter(new Predicate<Author>() {
            @Override
            public boolean test(Author author) {
                return author.getAge()>17;
            }
        }.or(new Predicate<Author>() {
```

```
            @Override
            public boolean test(Author author) {
                return author.getName().length()<2;
            }
        })).forEach(author ->
System.out.println(author.getName()));
```

例3：打印作家中年龄不大于17的作家。

```
//        打印作家中年龄不大于17的作家。
List<Author> authors = getAuthors();
authors.stream()
        .filter(new Predicate<Author>() {
            @Override
            public boolean test(Author author) {
                return author.getAge()>17;
            }
        }.negate()).forEach(author ->
System.out.println(author.getAge()));
```

# 六、方法引用【了解】

我们在使用lambda时，如果方法体中只有一个方法的调用的话（包括构造方法），我们可以用方法引用进一步简化代码。

我们在使用lambda时不需要考虑什么时候用方法引用，用哪种方法引用，方法引用的格式是什么。我们只需要在写完lambda方法发现方法体只有一行代码，并且是方法的调用时使用快捷键尝试是否能够转换成方法引用即可。

当我们方法引用使用的多了慢慢的也可以直接写出方法引用。

基本格式：

> 类名或者对象名::方法名

由基本格式可知，有的时候使用类名，有的时候使用对象名，那到底如何使用呢？分为几种情况。

## 第一种：引用类的静态方法

格式为：

> 类名::方法名

使用前提：**如果我们在重写方法的时候，方法体中只有一行代码，并且这行代码是调用了某个类的静态方法，并且我们把要重写的抽象方法中所有的参数都按照顺序传入了这个静态方法中，这个时候我们就可以引用类的静态方法。**

例：将List转为List类型的age列表，由于age本身是Integer类型，因此需要经过两层转换：List转为List转为List类型。

使用匿名内部类写法：

```
List<Author> authorList = getAuthors();
authorList.stream()
        .map(new Function<Author, Integer>() {
            @Override
            public Integer apply(Author author) {
                return author.getAge();
            }
        }).map(new Function<Integer, String>() {
            @Override
            public String apply(Integer age) {
                return String.valueOf(age);
            }
        });
```

如上，重写Function的apply函数时，方法体中只有一行代码，满足使用条件；这行代码是调用了String的静态方法valueOf，满足使用条件；最后一个条件需要满足：”把要重写的抽象方法中所有的参数都按照顺序传入了这个静态方法中“。

第一次重写方法，并没有将author对象传入方法的参数，因此不能引用类的静态方法；第二个重写方法，将age传入了方法参数，因此可以引用类的静态方法，简化后写法如下：

```
List<Author> authorList = getAuthors();
authorList.stream()
        .map(author -> author.getAge())
        .map(String::valueOf);
```

## 第二种：引用对象的实例方法

基本格式：

> 对象名::方法名

使用前提：**如果我们在重写方法的时候，方法体中只有一行代码，并且这行代码是调用了某个对象的成员方法，并且我们把要重写的抽象方法中所有的参数都按照顺序传入了这个成员方法中，这个时候我们就可以引用对象的实例方法。**

## 第三种：引用类的实例方法

格式为：

> 类名::方法名

使用前提：**如果我们在重写方法的时候，方法体中只有一行代码，并且这行代码是调用了第一个参数的成员方法，并且我们把要重写的抽象方法中剩余的所有的参数都按照顺序传入了这个成员方法中，这个时候我们就可以引用类的实例方法。**

第二种和第三种一起举例说明。

例子：

```
StringBuilder sb = new StringBuilder();
List<Author> authorList = getAuthors();
authorList.stream()
        .distinct()
        .map(new Function<Author, String>() {
            @Override
            public String apply(Author author) {
                return author.getName();
            }
        })
        .forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                sb.append(s);
            }
        });
System.out.println(sb.toString());
```

对于第一次重写方法，满足如下条件：

- 方法体中只有一行代码；
- 并且这行代码是调用了第一个参数的成员方法，这里是调用了author的getName()成员方法；
- 并且我们把要重写的抽象方法中剩余的所有的参数都按照顺序传入了这个成员方法中---名，没有剩余参数了，所以方法参数为空。

可以看到，满足第三种”引用类的实例方法“所有使用前提，我们可以进行简化。

对于第二次重写方法，满足如下条件：

- 方法体中只有一行代码；
- 并且这行代码是调用了某个对象的成员方法，sb对象调用了append成员方法；
- 并且我们把要重写的抽象方法中所有的参数都按照顺序传入了这个成员方法中---传递了s到append方法参数中。

可以看到，满足第二种”引用对象的实例方法“所有使用前提，我们也可以进行简化，
最终如下：

```
StringBuilder sb = new StringBuilder();
List<Author> authorList = getAuthors();
authorList.stream()
        .distinct()
        .map(Author::getName)//第三种：引用类的实例方法（类名::方法名）
        .forEach(sb::append);//第二种：引用对象的实例方法（对象名::方法
名）
System.out.println(sb.toString());
```

### 第四种：构造器引用

格式为：

```
类名::new
```

使用前提：**如果我们在重写方法的时候，方法体中只有一行代码，并且这行代码是调用了某个类的构造方法，并且我们把要重写的抽象方法中的所有的参数都按照顺序传入了这个构造方法中，这个时候我们就可以引用构造器。**

例如：

```
List<Author> authors = getAuthors();
authors.stream()
        .map(author -> author.getName())
        .map(name->new StringBuilder(name))
        .map(sb->sb.append("-第一个").toString())
        .forEach(str-> System.out.println(str));
```

优化后：

```
List<Author> authors = getAuthors();
authors.stream()
        .map(author -> author.getName())
        .map(StringBuilder::new)
        .map(sb->sb.append("-第一个").toString())
        .forEach(str-> System.out.println(str));
```

我们只需要大概了解有这么几种情况可以使用方法引用，不必太过记忆。

# 七、高级用法

## 基本数据类型优化

我们之前用到的很多Stream的方法由于都使用了泛型。所以涉及到的参数和返回值都是引用数据类型。

即使我们操作的是整数小数，但是实际用的都是他们的包装类。JDK5中引入的自动装箱和自动拆箱让我们在使用对应的包装类时就好像使用基本数据类型一样方便。但是你一定要知道装箱和拆箱肯定是要消耗时间的。虽然这个时间消耗很下。但是在大量的数据不断的重复装箱拆箱的时候，你就不能无视这个时间损耗了。

例如：

```
List<Author> authors = getAuthors();
authors.stream()
        .map(author -> author.getAge())
        .map(age -> age + 10)
        .filter(age->age>18)
        .map(age->age+2)
        .forEach(System.out::println);
```

第一步map获取的是List的age集合，第二个map对这个age集合数值加10，此时就会涉及到自动拆箱和装箱操作（将age转为int基本数据类型，加10操作，最后再转为Integer对象类型）。

所以为了让我们能够对这部分的时间消耗进行优化。Stream还提供了很多专门针对基本数据类型的方法。

例如：
mapToInt,mapToLong,mapToDouble,flatMapToInt,flatMapToDouble等。

```
authors.stream()
    .mapToInt(author -> author.getAge())
    .map(age -> age + 10)
    .filter(age->age>18)
    .map(age->age+2)
    .forEach(System.out::println);
```

## 并行流

当流中有大量元素时，我们可以使用并行流去提高操作的效率。其实并行流就是把任务分配给多个线程去完全。如果我们自己去用代码实现的话其实会非常的复杂，并且要求你对并发编程有足够的理解和认识。而如果我们使用Stream的话，我们只需要修改一个方法的调用就可以使用并行流来帮我们实现，从而提高效率。

先过滤再求和的一个小示例：

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
Integer sum = stream.filter(num -> num > 5)//过滤，排除小于5的
        .reduce((result, element) -> (result + element))//求和
        .get();
System.out.println(sum);//40
```

parallel方法可以把串行流转换成并行流。

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
Integer sum = stream.parallel()
        .peek(new Consumer<Integer>() {
            @Override
            public void accept(Integer num) {
                System.out.println(num+"----
"+Thread.currentThread().getName());
            }
        }).filter(num -> num > 5)
        .reduce((result, ele) -> result + ele)
        .get();
System.out.println(sum);
```

peek方法是stream专门提供的一个调测方法，只消费数据，不会影响数据，且可以在任何位置调用，不像forEach是终结操作，不可进行后续流处理。使用peek方法可以帮助我们判断是否真的使用了多线程。

打印出来的结果：

```
5----ForkJoinPool.commonPool-worker-6
2----ForkJoinPool.commonPool-worker-3
8----ForkJoinPool.commonPool-worker-6
10----ForkJoinPool.commonPool-worker-6
7----main
3----ForkJoinPool.commonPool-worker-1
9----ForkJoinPool.commonPool-worker-2
6----ForkJoinPool.commonPool-worker-4
4----ForkJoinPool.commonPool-worker-5
1----ForkJoinPool.commonPool-worker-7
40
```

如果不用parallel，那么都在main线程中执行：

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
Integer sum = stream
        .peek(new Consumer<Integer>() {
            @Override
            public void accept(Integer num) {
                System.out.println(num+"----
"+Thread.currentThread().getName());
            }
        }).filter(num -> num > 5)
        .reduce((result, ele) -> result + ele)
        .get();
System.out.println(sum);
```

执行结果：

```
1----main
2----main
3----main
4----main
5----main
6----main
7----main
8----main
9----main
10----main
40
```

也可以通过parallelStream直接获取并行流对象：

```
List<Author> authors = getAuthors();
authors.parallelStream()
        .map(author -> author.getAge())
        .map(age -> age + 10)
        .filter(age->age>18)
        .map(age->age+2)
        .forEach(System.out::println);
```

在数据量较大、多核的条件下，使用多线程来处理是可以大大提高执行效率的。