Java8 Stream 编码实战

作者：OKevin

公众号：CoderBuff

目录

写在前面的话

当我第一次在项目代码中看到Stream流的时候，心里不由得骂了一句“傻X”炫什么技。当我开始尝试在代码中使用Stream时，不由得感叹真香。

记得以前有朋友聊天说，他在代码中用了Lambda表达式结果CodeReview的时候老大让它改了。我在“第三章 Stream流”写了，“**简洁的后果就是，代码变得不那么好读，其实并不是代码的可读性降低了，而只是代码不是按照你的习惯去写的**”。人们不愿意去改变，因为改变似乎会给他带来风险，会让他在未知的领域不知所措，甚至被淘汰。所以人们开始找一些借口，可读性不高、炫技、难以调试、难以维护……

如果你从未了解过或者对Java8的Stream知之甚少，我建议可以从第一章开始慢慢品慢慢实践，如果你需要救急需要急切的知道怎么使List转换为Map结构等实际的场景，那么我建议你可以直接跳转到“第五章 Stream流编码实战”，我相信在第五章已经涵盖了大部分场景，也期待你能根据示例代码就能编写出符合自身业务的代码。如果有其他没有覆盖到的场景，我也非常期待你能通过公众号“CoderBuff”与我交流。

《Java8 Stream编码实战》的代码全部在

https://github.com/yu-linfeng/BlogRepositories/tree/master/repositories/stream-coding

一定要配合源码阅读，并且不断加以实践，才能更好的掌握Stream。

知识，是为了更好的编程。

OKevin

2020年3月18日夜晚 于成都

第一章 认识Java8以及函数式编程

尽管距离Java8发布已经过去7、8年的时间，但时至今日仍然有许多公司、项目停留在Java7甚至更早的版本。即使已经开始使用Java8的项目，大多数程序员也仍然采用“传统”的编码方式。

即使是在Java7就已经有了处理异常的新方式——`try-with-resources`，但大多数程序员也仍然采用在`finally`语句中关闭相应的资源。

我认为Java8和Java5的意义同等重要，Java5的众多新特性使得Java正式迈入编程界的统治地位。同样，Java8的发布，也使得这一门“古老”的语言具备了更加**现代化**的特性。

Java8最为引入瞩目就是支持**函数式编程**。

如果说面向对象编程是对数据的抽象，那么函数式编程就是对**行为的抽象**[1]。

```
button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.out.println("button clicked");
        }
});
```

以上示例来自于《Java 8函数式编程》

这个示例是为了一个按钮增加一个监听，当点击这个按钮时，将会触发打印“button clicked”行为。

在Java支持函数式编程以前，我们如果需要传递一个行为常用的方式就是传递一个对象，而**匿名内部类**正是为了方便将代码作为数据进行传递。

当然，函数式编程，并不是在Java8中才提出来的新概念，

函数式编程属于编程范式中的一种，它起源于一个数学问题。我们并不需要过多的了解函数式编程的历史，要追究它的历史以及函数式编程，关于范畴论、柯里化早就让人立马放弃学习函数式编程了。

对于函数式编程我们所要知道的是，它能将**一个行为传递作为参数进行传递**。至于其他的，就留给学院派吧。

---

1. 《On Java 8》 ↩

第二章 Lambda表达式

在第一章的示例中，我们看到在以前想要传递一个行为，我们通常使用的是**匿名内部类**，而从Java8开始，引入了一种全新更为简洁的方式来支持函数式编程，那就是——Lambda表达式。

我们把第一章中的示例改为Lambda作为本章的开始。

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Lambda表达式语法规则主体分为两个部分，中间用“->”右箭头连接，左边代表参数，右边代表函数主体。

2.1 函数式接口

在Java中有一个接口中只有一个方法表示某特定方法并反复使用，例如`Runnable`接口中只有`run`方法就表示执行的线程任务。

Java8中对于这样的接口有了一个特定的名称——**函数式接口**。Java8中即使是支持函数式编程，也并没有再标新立异另外一种语法表达。所以只要是只有一个方法的接口，都可以改写成Lambda表达式。在Java8中新增了`java.util.function`用来支持Java的函数式编程，其中的接口均是只包含一个方法。

例如`Predicate`接口中只包含`test`方法，该函数接口接受一个输入参数，返回一个布尔值。

函数式接口中的方法可以有参数、无参数、有返回值、无返回值。

- `() -> System.out.println("hellobug")`，表示无参数。
- `event -> System.out.println("hellobug")`，表示只有一个参数。
- `(x, y) -> {System.out.println(x); System.out.println(y);}`，表示两个参数，可以不必指定参数类型，为了更清楚地表达意图，最好还是加上参数类型，`(String x, String y) -> {System.out.println(x); System.out.println(y);}`。

接下来我们来编写一个带参数且有返回的**函数式接口**。

```
package com.coderbuff.chapter2_lambda.function;

/**
 * 函数式接口
 * @FunctionalInterface 注解只是为了表明这是一个函数式接口，函数式接口只能包含一个方法。
 * @author okevin
 * @date 2020/3/14 23:32
 */
@FunctionalInterface
public interface FunctionalInterfaceDemo {
    boolean test(Integer x);
}
```

com.coderbuff.chapter2_lambda.function.FunctionalInterfaceDemo

> 除了@FunctionalInterface注解，其它和一个普通的接口无任何差别。@FunctionalInterface注解只是为了标注这是一个函数式接口，如果标注了@FunctionalInterface注解，此时接口中就只能包含一个方法，因为函数式接口只能包含一个方法。

接着我们在测试类中编写一个方法，方法的参数就是这个函数式接口，这代表了我们将传递**行为**。

```
package com.coderbuff.chapter2_lambda.function;

/**
 * 按匿名类的方式使用一个函数式接口，传递行为
 * @author okevin
 * @date 2020/3/14 23:42
 */
public class AnonymousInnerClassTest {

    private void testAnonymousInnerClass(FunctionalInterfaceDemo
functionalInterfaceDemo) {
        Integer number = 1;
        boolean result = functionalInterfaceDemo.test(number);
        System.out.println(result);
    }
}
```

com.coderbuff.chapter2_lambda.function.AnonymousInnerClassTest

`testAnonymousInnerClass`方法的含义表示将通过`FunctionalInterfaceDemo#test`方法判断传入的参数1返回布尔值。

我们应该如何通过Lambda表达式来使用这个函数式接口呢?

前面我们说了，这个参数代表了我们将传递一个**行为**，这个行为决定了1返回是true还是false，我们先通过**匿名内部类**实现这个接口。

```
package com.coderbuff.chapter2_lambda.function;
```

```java
/**
 * 按匿名类的方式使用一个函数式接口，传递行为
 * @author okevin
 * @date 2020/3/14 23:42
 */
public class AnonymousInnerClassTest {

    private void testAnonymousInnerClass(FunctionalInterfaceDemo
functionalInterfaceDemo) {
        Integer number = 1;
        boolean result = functionalInterfaceDemo.test(number);
        System.out.println(result);
    }

    public static void main(String[] args) {
        AnonymousInnerClassTest anonymousInnerClassTest = new
AnonymousInnerClassTest();

        anonymousInnerClassTest.testAnonymousInnerClass(new
FunctionalInterfaceDemo() {
            @Override
            public boolean test(Integer x) {
                if (x > 1) {
                    return true;
                }
                return false;
            }
        });
    }
}
```


这是在Java8之前通过匿名内部类实现**行为的传递**，在有了Lambda表达式后，通过上文的Lambda表达式语法规则，这是一个参数+一个返回（Lambda表达式中有返回值时return可以省略），并且有多行代码。

```java
anonymousInnerClassTest.testAnonymousInnerClass(number -> {
    if (number > 1) {
        return true;
    }
    return false;
});
```

第三章 Stream流

对于初学者，必须要声明一点的是，Java8中的Stream尽管被称作为“流”，但它和文件流、字符流、字节流**完全没有任何关系**。Stream流使程序员得以站在更高的抽象层次上对集合进行操作[1]。也就是说Java8中新引入的Stream流是针对集合的操作。

3.1 迭代

我们在使用集合时，最常用的就是迭代。

```
public int calcSum(List<Integer> list) {
    int sum = 0;
    for (int i = 0; i < list.size(); i++) {
        sum += list.get(i);
    }
    return sum;
}
```

com.coderbuff.chapter3_stream.chapter3_1.ForDemo#calcSum

例如，我们可能会对集合中的元素累加并返回结果。这段代码由于for循环的样板代码并不能很清晰的传达程序员的意图。也就是说，实际上除了方法名叫“计算总和”，程序员必须阅读整个循环体才能理解。你可能觉得一眼就能理解上述代码的意图，但如果碰上下面的代码，你还能一眼理解吗？


```
public Map<Long, List<Student>> useFor(List<Student> students) {
    Map<Long, List<Student>> map = new HashMap<>();
    for (Student student : students) {
        List<Student> list = map.get(student.getStudentNumber());
        if (list == null) {
            list = new ArrayList<>();
            map.put(student.getStudentNumber(), list);
        }
        list.add(student);
    }
    return map;
}
```

阅读完这个循环体以及包含的if判断条件，大概可以知道这是想使用“studentNumber”对“Student”对象分组。这段代码在Stream进行重构后，将会变得非常简洁和**易读**。

```
public Map<Long, List<Student>> useStreamByGroup(List<Student> students) {
    Map<Long, List<Student>> map =
students.stream().collect(Collectors.groupingBy(Student::getStudentNumber));
    return map;
}
```

当第一次看到这样的写法时，可能会认为这样的代码可读性不高，不容易测试。我相信，当你在学习掌握Stream后会重新改变对它的看法。

3.2 Stream

3.2.1 创建

要想使用Stream，首先要创建一个流，创建流最常用的方式是直接调用集合的`stream`方法。

```
/**
 * 通过集合构造流
 */
private void createByCollection() {
    List<Integer> list = new ArrayList<>();
    Stream<Integer> stream = list.stream();
}
```

com.coderbuff.chapter3_stream.chapter3_2.StreamCreator#createByCollection

也能通过数组构造一个流。

```
/**
 * 通过数组构造流
 */
private void createByArrays() {
    Integer[] intArrays = {1, 2, 3};
    Stream<Integer> stream = Stream.of(intArrays);
    Stream<Integer> stream1 = Arrays.stream(intArrays);
}
```

com.coderbuff.chapter3_stream.chapter3_2.StreamCreator#createByArrays

学习Stream流，掌握集合创建流就足够了。

3.2.2 使用

对于Stream流操作共分为两个大类：**惰性求值**、**及时求值**。

所谓惰性求值，指的是操作最终不会产生新的集合。及时求值，指的是操作会产生新的集合。举以下示例加以说明：

```
/**
```

```
 * 通过for循环过滤元素返回新的集合
 * @param list 待过滤的集合
 * @return 过滤后的集合
 */
private List<Integer> filterByFor(List<Integer> list) {
    List<Integer> filterList = new ArrayList<>();

    for (Integer number : list) {
        if (number > 1) {
            filterList.add(number);
        }
    }
    return filterList;
}
```

com.coderbuff.chapter3_stream.chapter3_3.Example#filterByFor

通过for循环过滤元素返回新的集合，这里的“过滤”表示排除不符合条件的元素。我们使用Stream流过滤并返回新的集合：

```
/**
 * 通过Stream流过滤元素返回新的集合
 * @param list 待过滤的集合
 * @return 新的集合
 */
private List<Integer> filterByStream(List<Integer> list) {
    return list.stream()
            .filter(number -> number > 1)
            .collect(Collectors.toList());
}
```

com.coderbuff.chapter3_stream.chapter3_3.Example#filterByStream

Stream操作时，先调用了`filter`方法传入了一个Lambda表达式代表过滤规则，后调用了`collect`方法表示将流转换为List集合。

按照常理来想，一个方法调用完后，接着又调用了一个方法，看起来好像做了两次循环，把问题搞得更复杂了。但实际上，这里的`filter`操作是**惰性求值**，它并不会返回新的集合，这就是Stream流设计精妙的地方。既能在保证可读性的同时，也能保证性能不会受太大影响。

所以使用Stream流的理想方式就是，**形成一个惰性求值的链，最后用一个及早求值的操作返回想要的结果。**

我们不需要去记哪些方法是惰性求值，如果方法的返回值是Stream那么它代表的就是惰性求值。如果返回另外一个值或空，那么它代表的就是及早求值。

3.2.3 常用的Stream操作

map

map操作不好理解，它很容易让人以为这是一个转换为Map数据结构的操作。实际上他是将集合中的元素类型，转换为另外一种数据类型。

例如，你想将“学生”类型的集合转换为只有“学号”类型的集合，应该怎么做?

```
/**
 * 通过for循环提取学生学号集合
 * @param list 学生对象集合
 * @return 学生学号集合
 */
public List<Long> fetchStudentNumbersByFor(List<Student> list) {
    List<Long> numbers = new ArrayList<>();
    for (Student student : list) {
        numbers.add(student.getStudentNumber());
    }
    return numbers;
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamMapDemo#fetchStudentNumbersByFor

这是只借助JDK的“传统”方式。如果使用Stream则可以直接通过`map`操作来获取只包含学生学号的集合。

```
/**
 * 通过Stream map提取学生学号集合
 * @param list 学生对象集合
 * @return 学生学号集合
 */
public List<Long> fetchStudentNumbersByStreamMap(List<Student> list) {
    return list.stream()
               .map(Student::getStudentNumber)
               .collect(Collectors.toList());
}
```


com.coderbuff.chapter3_stream.chapter3_4.StreamMapDemo#fetchStudentNumbersByStreamMap

`map`传入的是一个方法，同样可以理解为传入的是一个“行为”，在这里我们传入方法“getStudentNumber”表示将通过这个方法进行转换分类。

> “Student::getStudentNumber”叫**方法引用**，它是“student -> student.getStudentNumber()”的简写。表示**直接引用已有Java类或对象的方法或构造器**。在这里我们是需要传入“getStudentNumber”方法，在有的地方，你可能会看到这样的代码“Student::new”，new调用的就是构造方法，表示创建一个对象。方法引用，可以将我们的代码变得更加紧凑简洁。

我们再举一个例子，将小写的字符串集合转换为大写字符串集合。

```java
/**
 * 通过Stream map操作将小写的字符串集合转换为大写
 * @param list 小写字符串集合
 * @return 大写字符串集合
 */
public List<String> toUpperByStreamMap(List<String> list) {
    return list.stream()
               .map(String::toUpperCase)
               .collect(Collectors.toList());
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamMapDemo#toUpperByStreamMap

filter

`filter`，过滤。这里的过滤含义是“排除不符合某个条件的元素”，也就是返回true的时候保留，返回false排除。

我们仍然以“学生”对象为例，要排除掉分数低于60分的学生。

```java
/**
 * 通过for循环筛选出分数大于60分的学生集合
 * @param students 待过滤的学生集合
 * @return 分数大于60分的学生集合
 */
public List<Student> fetchPassedStudentsByFor(List<Student> students) {
    List<Student> passedStudents = new ArrayList<>();
    for (Student student : students) {
        if (student.getScore().compareTo(60.0) >= 0) {
            passedStudents.add(student);
        }
    }
    return passedStudents;
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamFilterDemo#fetchPassedStudentsByFor

这是我们通常的实现方式，通过for循环能解决“一切”问题，如果使用Stream filter一行就搞定。

```
/**
 * 通过Stream filter筛选出分数大于60分的学生集合
 * @param students 待过滤的学生集合
 * @return 分数大于60分的学生集合
 */
public List<Student> fetchPassedStudentsByStreamFilter(List<Student> students)
{
    return students.stream()
            .filter(student -> student.getScore().compareTo(60.0) >= 0)
            .collect(Collectors.toList());
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamFilterDemo#fetchPassedStudentsByStreamFilter

sorted

排序，也是日常最常用的操作之一。我们常常会把数据按照修改或者创建时间的倒序、升序排列，这步操作通常会放到SQL语句中。但如果实在是遇到要对集合进行排序时，我们通常也会使用`Comparator.sort`静态方法进行排序，如果是复杂的对象排序，还需要实现`Comparator`接口。

```
/**
 * 通过Collections.sort静态方法 + Comparator匿名内部类对学生成绩进行排序
 * @param students 待排序学生集合
 * @return 排好序的学生集合
 */
private List<Student> sortedByComparator(List<Student> students) {
    Collections.sort(students, new Comparator<Student>() {
        @Override
        public int compare(Student student1, Student student2) {
            return student1.getScore().compareTo(student2.getScore());
        }
    });
    return students;
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamSortedDemo#sortedByComparator

> 关于`Comparator`可以查看这篇文章《似懂非懂的Comparable与Comparator》。简单来讲，我们需要实现`Compartor`接口的`compare`方法，这个方法有两个参数用于比较，返回1代表前者大于后者，返回0代表前者等于后者，返回-1代表前者小于后者。

当然我们也可以手动实现冒泡算法对学生成绩进行排序，不过这样的代码大多出现在课堂教学中。

```
/**
 * 使用冒泡排序算法对学生成绩进行排序
 * @param students 待排序学生集合
 * @return 排好序的学生集合
 */
private List<Student> sortedByFor(List<Student> students) {
```

```
    for (int i = 0; i < students.size() - 1; i++) {
        for (int j = 0; j < students.size() - 1 - i; j++) {
            if (students.get(j).getScore().compareTo(students.get(j +
1).getScore()) > 0) {
                Student temp = students.get(j);
                students.set(j, students.get(j + 1));
                students.set(j + 1, temp);
            }
        }
    }
    return students;
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamSortedDemo#sortedByFor

在使用Stream sorted后，你会发现代码将变得无比简洁。

```
/**
 * 通过Stream sorted对学生成绩进行排序
 * @param students 待排序学生集合
 * @return 排好序的学生集合
 */
private List<Student> sortedByStreamSorted(List<Student> students) {
    return students.stream()
                   .sorted(Comparator.comparing(Student::getScore))
                   .collect(Collectors.toList());
}
```


com.coderbuff.chapter3_stream.chapter3_4.StreamSortedDemo#sortedByStreamSorted

简洁的后果就是，代码变得不那么好读，其实并不是代码的可读性降低了，而只是代码不是按照你的习惯去写的。而大部分人恰好只习惯墨守成规，而不愿意接受新鲜事物。

上面的排序是按照从小到大排序，如果想要从大到小应该如何修改呢?

`Compartor.sort`方法和for循环调换if参数的位置即可。

```
return student1.getScore().compareTo(student2.getScore());
修改为
return student2.getScore().compareTo(student1.getScore());
```

```
if (students.get(j).getScore().compareTo(students.get(j + 1).getScore()) > 0)
修改为
if (students.get(j).getScore().compareTo(students.get(j + 1).getScore()) < 0)
```

这改动看起来很简单，但如果这是一段**没有注释并且不是你本人写的代码**，你能一眼知道是按降序还是升序排列吗？你还能说这是可读性强的代码吗?

如果是Stream操作。

```
return students.stream()
                .sorted(Comparator.comparing(Student::getScore))
                .collect(Collectors.toList());
修改为
return students.stream()
                .sorted(Comparator.comparing(Student::getScore).reversed())
                .collect(Collectors.toList());
```

这就是**声明式编程**，你只管叫它做什么，而不像**命令式编程**叫它如何做。

reduce

`reduce`是将传入一组值，根据计算模型输出一个值。例如求一组值的最大值、最小值、和等等。

不过使用和读懂`reduce`还是比较晦涩，如果是简单最大值、最小值、求和计算，Stream已经为我们提供了更简单的方法。如果是复杂的计算，可能为了代码的可读性和维护性还是建议用传统的方式表达。

我们来看几个使用`reduce`进行累加例子。

```
/**
 * Optional<T> reduce(BinaryOperator<T> accumulator);
 * 使用没有初始值对集合中的元素进行累加
 * @param numbers 集合元素
 * @return 累加结果
 */
private Integer calcTotal(List<Integer> numbers) {
    return numbers.stream()
            .reduce((total, number) -> total + number).get();
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamReduceDemo#calcTotal

`reduce`有3个重载方法， 第一个例子调用的是`Optional<T> reduce(BinaryOperator<T> accumulator);`它只有`BinaryOperator`一个参数，这个接口是一个**函数接口**，代表它可以接收一个Lambda表达式，它继承自`BiFunction`函数接口，在`BiFunction`接口中，只有一个方法：

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

这个方法有两个参数。也就是说，传入`reduce`的Lambda表达式需要“实现”这个方法。如果不理解这是什么意思，我们可以抛开Lambda表达式，从纯粹传统的接口角度去理解。

首先，`Optional<T> reduce(BinaryOperator<T> accumulator);`方法接收`BinaryOperator`类型的对象，而`BinaryOperator`是一个接口并且继承自`BiFunction`接口，而在`BiFunction`中只有一个方法定义`R apply(T t, U u)`，也就是说我们需要实现`apply`方法。

其次，接口需要被实现，我们不妨传入一个匿名内部类，并且实现`apply`方法。

```
private Integer calcTotal(List<Integer> numbers) {
    return numbers.stream()
            .reduce(new BinaryOperator<Integer>() {
                @Override
                public Integer apply(Integer integer, Integer integer2) {
                    return integer + integer2;
                }
            }).get();
}
```

最后，我们在将匿名内部类改写为Lambda风格的代码，箭头左边是参数，右边是函数主体。

```
private Integer calcTotal(List<Integer> numbers) {
    return numbers.stream()
            .reduce((total, number) -> total + number).get();
}
```

至于为什么两个参数相加最后就是不断累加的结果，这就是`reduce`的内部实现了。

接着看第二个例子：

```
/**
 * T reduce(T identity, BinaryOperator<T> accumulator);
 * 赋初始值为1，对集合中的元素进行累加
 * @param numbers 集合元素
 * @return 累加结果
 */
private Integer calcTotal2(List<Integer> numbers) {
    return numbers.stream()
            .reduce(1, (total, number) -> total + number);
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamReduceDemo#calcTotal2

第二个例子调用的是`reduce`的`T reduce(T identity, BinaryOperator<T> accumulator);`重载方法，相比于第一个例子，它多了一个参数“identity”，这是进行后续计算的初始值，`BinaryOperator`和第一个例子一样。

第三个例子稍微复杂一点，前面两个例子集合中的元素都是基本类型，而现实情况是，集合中的参数往往是一个**对象**我们常常需要对对象中的某个字段做累加计算，比如计算学生对象的总成绩。

我们先来看for循环怎么做的：

```
/**
 * 通过for循环对集合中的学生成绩字段进行累加
 * @param students 学生集合
 * @return 分数总和
 */
private Double calcTotalScoreByFor(List<Student> students) {
    double total = 0;
    for (Student student : students) {
        total += student.getScore();
    }
    return total;
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamReduceDemo#calcTotalScoreByFor

要按前文的说法，“这样的代码充斥了样板代码，除了方法名，代码并不能直观的反应程序员的意图，程序员需要读完整个循环体才能理解”，但凡事不是绝对的，如果换做`reduce`操作：

```
/**
 * <U> U reduce(U identity,
 *                  BiFunction<U, ? super T, U> accumulator,
 *                  BinaryOperator<U> combiner);
 * 集合中的元素是"学生"对象，对学生的"score"分数字段进行累加
 * @param students 学生集合
 * @return 分数总和
 */
private Double calcTotalScoreByStreamReduce(List<Student> students) {
    return students.stream()
            .reduce(Double.valueOf(0),
                    (total, student) -> total + student.getScore(),
                    (aDouble, aDouble2) -> aDouble + aDouble2);
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamReduceDemo#calcTotalScoreByStreamReduce

这样的代码，已经不是样板代码的问题了，是大部分程序员即使读十遍可能也不知道要表达什么含义。但是为了学习Stream我们还是要硬着头皮去理解它。

Lambda表达式不好理解，过于简洁的语法，也代表更少的信息量，我们还是先将Lambda表达式还原成匿名内部类。

```
private Double calcTotalScoreByStreamReduce(List<Student> students) {
    return students.stream()
            .reduce(Double.valueOf(0), new BiFunction<Double, Student, Double>
() {
                @Override
                public Double apply(Double total, Student student) {
                    return total + student.getScore();
                }
```

```
            }, new BinaryOperator<Double>() {
                @Override
                public Double apply(Double aDouble, Double aDouble2) {
                    return aDouble + aDouble2;
                }
            });
}
```

`reduce`的第三个重载方法`<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);`一共有3个参数，与第一、二个重载方法不同的是，第一、第二个重载方法参数和返回类型都是泛型"T"，意思是入参和返回都是同一种数据类型。但在第三个例子中，入参是`Student`对象，返回却是`Double`，显然不能使用第一、二个重载方法。

第三个重载方法的第一个参数类型是泛型"U"，它的返回类型也是泛型"U"，所以第一个参数类型，代表了返回的数据类型，我们必须将第一个类型定义为`Double`，**例子中的入参是`Double.valueOf(0)`表示了累加的初始值为0，且返回值是`Double`类型**。第二个参数可以简单理解为"应该如何计算，累加还是累乘"的计算模型。最难理解的是第三个参数，因为前两个参数类型看起来已经能满足我们的需求，为什么还有第三个参数呢?

当我在第三个参数中加上一句输出时，发现它确实没有用。

```
private Double calcTotalScoreByStreamReduce(List<Student> students) {
    return students.stream()
            .reduce(Double.valueOf(0), new BiFunction<Double, Student, Double>
() {
                @Override
                public Double apply(Double total, Student student) {
                    return total + student.getScore();
                }
            }, new BinaryOperator<Double>() {
                @Override
                public Double apply(Double aDouble, Double aDouble2) {
                    System.out.println("第三个参数的作用");
                    return aDouble + aDouble2;
                }
            });
}
```

控制台没有输出"第三个参数的作用"，改变它的返回值最终结果也没有任何改变，这的确表示它**真的没有用**。

第三个参数在这里的确没有用，这是因为我们目前所使用的Stream流是串行操作，它在**并行Stream流**中发挥的是**多路合并**的作用，在下一章会继续介绍并行Stream流，这里就不再多做介绍。

对于`reduce`操作，我的个人看法是，**不建议在现实中使用**。如果你有累加、求最大值、最小值的需求，Stream封装了更简单的方法。如果是特殊的计算，不如直接按for循环实现，如果一定要使用Stream对学生成绩求和也不妨换一个思路。

前面提到`map`方法可以将集合中的元素类型转换为另一种类型，那我们就能把学生的集合转换为分数的集合，再调用`reduce`的第一个重载方法计算总和：

```
/**
 * 先使用map将学生集合转换为分数的集合
 * 再使用reduce调用第一个重载方法计算总和
 * @param students 学生集合
 * @return 分数总和
 */
private Double calcTotalScoreByStreamMapReduce(List<Student> students) {
    return students.stream()
            .map(Student::getScore)
            .reduce((total, score) -> total + score).get();
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamReduceDemo#calcTotalScoreByStreamMapReduce

min

`min`方法能返回集合中的最小值。它接收一个`Comparator`对象，Java8对`Comparator`接口提供了新的静态方法`comparing`，这个方法返回`Comparator`对象，以前我们需要手动实现`compare`比较，现在我们只需要调用`Comparator.comparing`静态方法即可。

```
/**
 * 通过Stream min计算集合中的最小值
 * @param numbers 集合
 * @return 最小值
 */
private Integer minByStreamMin(List<Integer> numbers) {
    return numbers.stream()
                .min(Comparator.comparingInt(Integer::intValue)).get();
}
```


com.coderbuff.chapter3_stream.chapter3_4.StreamMinDemo#minByStreamMin

`Comparator.comparingInt`用于比较int类型数据。因为集合中的元素是Integer类型，所以我们传入Integer类型的iniValue方法。如果集合中是对象类型，我们直接调用`Comparator.comparing`即可。

```
/**
 * 通过Stream min计算学生集合中的最低成绩
 * @param students 学生集合
 * @return 最低成绩
 */
private Double minScoreByStreamMin(List<Student> students) {
    Student minScoreStudent = students.stream()
            .min(Comparator.comparing(Student::getScore)).get();
    return minScoreStudent.getScore();
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamMinDemo#minScoreByStreamMin

max

和`min`的用法相同，含义相反取最大值。这里不再举例。

summaryStatistics

求和操作也是常用的操作，利用`reduce`会让代码晦涩难懂，特别是复杂的对象类型。

好在Streaam提供了求和计算的简便方法——`summaryStatistics`，这个方法并不是Stream对象提供，而是`IntStream`，可以把它当做处理基本类型的流，同理还有`LongStream`、`DoubleStream`。

`summaryStatistics`方法也不光是只能求和，它还能求最小值、最大值。

例如我们求学生成绩的平均分、总分、最高分、最低分。

```
/**
 * 学生类型的集合常用计算
 * @param students 学生
 */
private void calc(List<Student> students) {
    DoubleSummaryStatistics summaryStatistics = students.stream()
            .mapToDouble(Student::getScore)
            .summaryStatistics();
    System.out.println("平均分：" + summaryStatistics.getAverage());
    System.out.println("总分：" + summaryStatistics.getSum());
    System.out.println("最高分：" + summaryStatistics.getMax());
    System.out.println("最低分：" + summaryStatistics.getMin());
}
```


com.coderbuff.chapter3_stream.chapter3_4.StreamSummaryStatisticsDemo#calc

返回的`summaryStatistics`包含了我们想要的所有结果，不需要我们单独计算。`mapToDouble`方法将Stream流按“成绩”字段组合成新的`DoubleStream`流，`summaryStatistics`方法返回的`DoubleSummaryStatistics`对象为我们提供了常用的计算。

灵活运用好`summaryStatistics`，一定能给你带来更少的bug和更高效的编码。

3.3 Collectors

前面的大部分操作都是以`collect(Collectors.toList())`结尾，看多了自然也大概猜得到它是将流转换为集合对象。最大的功劳当属Java8新提供的类——`Collectors`收集器。

`Collectors`不但有`toList`方法能将流转换为集合，还包括`toMap`转换为Map数据类型，还能**分组**。

```
/**
 * 将学生类型的集合转换为只包含名字的集合
 * @param students 学生集合
 * @return 学生姓名集合
 */
private List<String> translateNames(List<Student> students) {

    return students.stream()
                   .map(Student::getStudentName)
                   .collect(Collectors.toList());
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamCollectorsDemo#translateNames

```
/**
 * 将学生类型的集合转换为Map类型，key=学号，value=学生
 * @param students 学生集合
 * @return 学生Map
 */
private Map<Long, Student> translateStudentMap(List<Student> students) {
    return students.stream()
            .collect(Collectors.toMap(Student::getStudentNumber, student ->
student));
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamCollectorsDemo#translateStudentMap


```
/**
 * 按学生的学号对学生集合进行分组返回Map，key=学生学号，value=学生集合
 * @param students 学生集合
 * @return 按学号分组的Map
 */
private Map<Long, List<Student>> studentGroupByStudentNumber(List<Student>
students) {
    return students.stream()
            .collect(Collectors.groupingBy(Student::getStudentNumber));
}
```

com.coderbuff.chapter3_stream.chapter3_4.StreamCollectorsDemo#studentGroupByStudentNumber

这些是比较常见的场景，更加贴近实战的场景可直接跳转到“第五章 Stream流编码实战”。

---

1. 《Java 8函数式编程》↩

第四章 并行化Stream流

在现实当中，并行化流开始并没有引起我的注意，直到我发现了它的应用场景后才发现，并行化流在提高性能以及编码难易程度上，代码bug上似乎要更胜一筹。

“第三章 Stream流”一直介绍的是**串行化**的流，串行化的流如果你有心可以和for循环对比，会发现串行化的流在性能上是比for循环要差的。这也是部分人“鄙视”Stream流的一点。

4.1 并行与并发

并行，指的是**在同一时刻**多个任务同时执行。

并发，指的是**在同一时间段**多个任务交替执行。

当然，并行的执行速度更快，但并行也依赖硬件设置，因为它依赖硬件CPU是多核的场景。并发则不受限制。

parallelStream

想要把串行流转换为并行流很简单，只需要将`stream`修改为`parallelStream`，其它操作不变。

```
public void parallel(List<Student> students) {
    students.parallelStream()
            .map(Student::getStudentNumber)
            .collect(Collectors.toList());
}
```

com.coderbuff.chapter4_parallelstream.ParallelStreamDemo#parallel

Stream流的并行化操作，是一种**数据并行化**，流本身就擅长对数据进行运算。

当然并不是将代码里所有的串行流改为并行流就万事大吉性能翻倍了，数据少了不行，CPU核数不够也不行。所以要想真正能提高性能，还要针对实际请做测试才能得出结论。

我们分别举几个数据量不同的例子，来说明for循环、串行化Stream流、并行化Stream流的性能在我本机的性能。

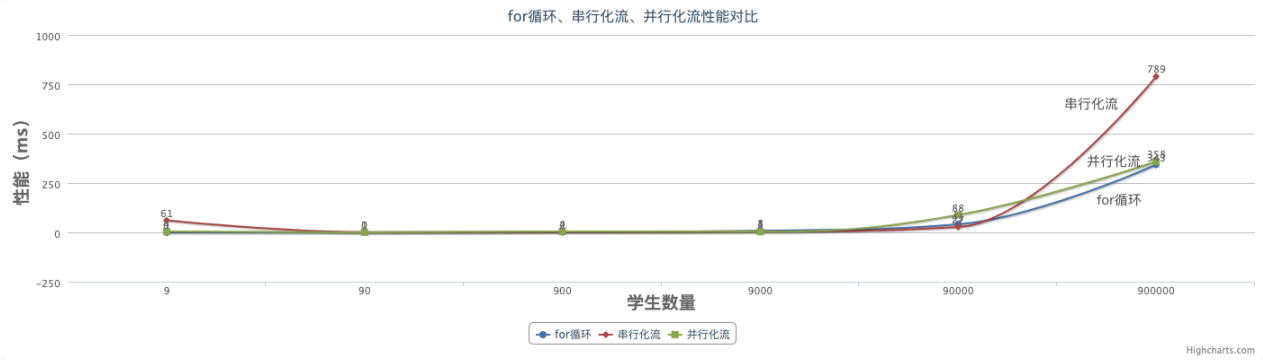
硬件概览：

型号名称：	MacBook Pro
型号标识符：	MacBookPro14,1
处理器名称：	Intel Core i5
处理器速度：	2.3 GHz
处理器数目：	1
核总数：	2
L2 缓存（每个核）：	256 KB
L3 缓存：	4 MB
超线程技术：	已启用
内存：	16 GB
Boot ROM 版本：	190.0.0.0.0
SMC 版本（系统）：	2.43f6
序列号（系统）：	[redacted]
硬件 UUID：	[redacted]

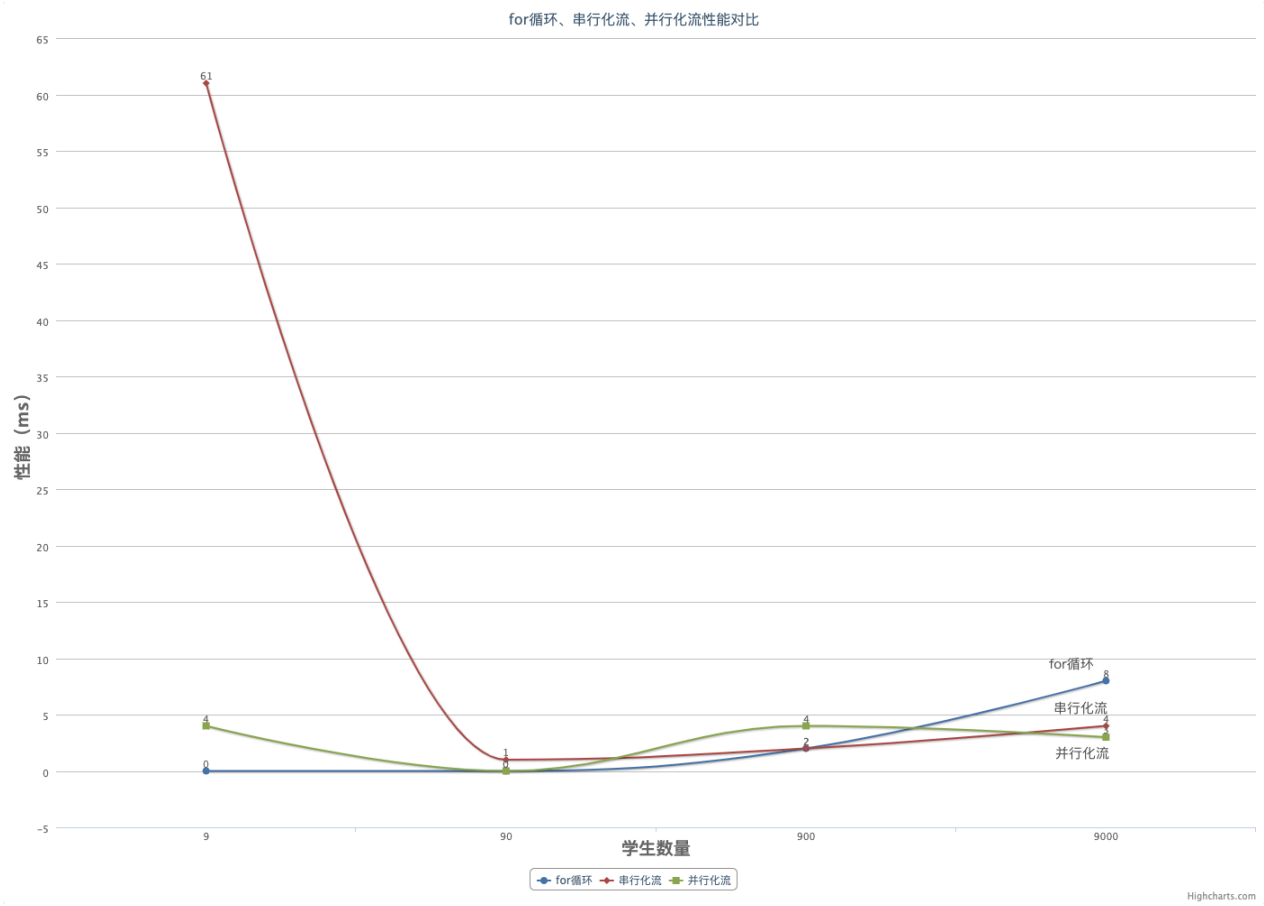
我测试了9个学生、90个学生、9000个学生、90000个学生后，三者的性能表现如下图所示：

`com.coderbuff.chapter4_parallelstream.PerformanceTests`

操作	9个学生	90个学生	900个学生	9000个学生	90000个学生	900000个学生
for循环	0ms	0ms	2ms	8ms	41ms	343ms
串行化Stream	61ms	1ms	2ms	4ms	27ms	789ms
并行化Stream	4ms	0ms	4ms	3ms	88ms	358ms



从曲线图可以看出90000个学生以前3者的性能都是几毫秒，并没有太大区别，从90000个学生过后，串行化流性能主键走弱，并行化流的性能开始逐渐赶上for循环，但注意**这并不意味着从900000个数据后并行化的数据就一定会超越for循环**。由于后两组测试数据比较大，我们去掉90000和900000这两组数据的性能曲线图。



从这张图可以看到，串行化流在数据量很小的情况下，性能最差。而并行化流则处于波动的状态。

所以单单从数据量上可以看出：

for循环的性能随着数据量的增加性能也越来越差。

串行化流则在数据量小的情况下性能差，数据量中、大的时候性能略高于for循环，但当数据量特别大时，性能也变得越差。

并行化流受CPU核数的影响，在本机2核下，在数据量小的情况下性能略高于串行化流，略低于for循环，在数据量中的情况下差不多，在数据量比较大时性能最差，但当数据量特别大时，性能也变得更好。

如果想要使用`parallelStream`想提高性能，一定要根据实际情况做好测试，因为**并行化的流性能不一定比串行化流性能高**。

第五章 Stream流编码实战

本章将尽可能涵盖常见的Stream应用场景，不再对比for循环实现方式，目标是希望你能根据示例代码就能编写出符合自身业务的代码。

5.1 冗余的学生成绩信息

5.1.1 数据

> 这里有一些学生课程成绩的数据，包含了学号、姓名、科目和成绩，一个学生会包含多条不同科目的数据。

ID	学号	姓名	科目	成绩
1	20200001	Kevin	语文	90
2	20200002	张三	语文	91
3	20200001	Kevin	数学	99
4	20200003	李四	语文	76
5	20200003	李四	数学	71
6	20200001	Kevin	英语	68
7	20200002	张三	数学	88
8	20200003	张三	英语	87
9	20200002	李四	英语	60

5.1.2 实战

场景一：通过学号，计算一共有多少个学生。

示例数据对于学生的学号和姓名是冗余的，要求按照学号去重。

```
/**
 * map 按学生的学号转换为新的数据结构类型
 * distinct 去重
 * count 计数
 * @param students 学生
 * @return 去重后的学生数量
 */
private long calcStudentNumber(List<Student> students) {
    return students.stream()
            .map(Student::getStudentNumber)
            .distinct()
            .count();
}
```

com.coderbuff.chapter5_action.chapter5_1.Scene1

场景二：通过学号+姓名，计算一共有多少个学生。

```
/**
 * map 按学生的学号+姓名转换为新的数据结构类型
 * distinct 去重
 * count 计数
 * @param students 学生
 * @return 去重后的学生数量
 */
private long calcStudentNumber(List<Student> students) {
    return students.stream()
            .map(student -> student.getStudentNumber() +
student.getStudentName())
            .distinct()
            .count();
}
```

com.coderbuff.chapter5_action.chapter5_1.Scene2

场景三：将学生List转换为Map类型，key=学号，value=姓名。

```
/**
 * collect 流->集合
 * Collectors.toMap 转换为Map类型
 * @param students 学生集合
 * @return key=学号, value=姓名
 */
private Map<Long, String> translateMap(List<Student> students) {
    return students.stream()
            .collect(Collectors.toMap(Student::getStudentNumber,
Student::getStudentName, (studentNumber1, studentNumber2) -> studentNumber2));
}
```

com.coderbuff.chapter5_action.chapter5_1.Scene3

> 在本示例数据中，学生的学号有重复，不能直接调用`students.stream().collect(Collectors.toMap(Student::getStudentNumber, Student::getStudentName)`，否则会报“Duplicate key”键值重复错误。第三个参数代表的是键值重复后的合并方式。

场景四：将学生List转换为Map类型，key=学号，value=学生。

```
/**
 * collect 流->集合
 * Collectors.toMap 转换为Map类型
 * @param students 学生集合
 * @return key=学号，value=学生信息
 */
private Map<Long, Student> translateMap(List<Student> students) {
    return students.stream()
            .collect(Collectors.toMap(Student::getStudentNumber, student ->
student, (studentNumber1, studentNumber2) -> studentNumber2));
}
```

com.coderbuff.chapter5_action.chapter5_1.Scene4

场景五：将学生List按学号相同的进行分组，key=学号，value=学生集合。


```
/**
 * collect 流->集合
 * Collectors.groupingBy 分组
 * @param students 学生信息
 * @return 按学号分组Map
 */
private Map<Long, List<Student>> groupingStudentByStudentNumber(List<Student>
students) {
    return students.stream()
            .collect(Collectors.groupingBy(Student::getStudentNumber));
}
```

com.coderbuff.chapter5_action.chapter5_1.Scene5

场景六：将学生List按学号相同的进行分组，key=学号，value=学生集合，同时key键按从小到大字典序排序。

```
/**
 * collect 流 -> 集合
 * Collectors.groupingBy 分组，第一个参数表示key键，第二个参数表示返回类型，value值按List结合收集
 * @param students 学生信息
 * @return key=学号，value=学生集合，key键按字典序
 */
private TreeMap<Long, List<Student>>
groupingStudentByStudentNumber(List<Student> students) {
    return students.stream()
            .collect(Collectors.groupingBy(Student::getStudentNumber,
TreeMap::new, Collectors.toList()));
}
```

com.coderbuff.chapter5_action.chapter5_1.Scene6

场景七：计算每个学生的总成绩（语文+数学+英语），并按Map结构返回，key=学号，value=总成绩。

```
/**
 * collect 流->集合
 * Collectors.groupingBy 分组
 * Collectors.summingDouble double类型数据求和
 * @param students 学生信息
 * @return key=学号，value=总成绩。
 */
private Map<Long, Double> calcTotalScore(List<Student> students) {
    return students.stream()
            .collect(Collectors.groupingBy(Student::getStudentNumber,
Collectors.summingDouble(Student::getScore)));

}
```

com.coderbuff.chapter5_action.chapter5_1.Scene7

场景八：学生的语文成绩乘以30%，数学成绩50%，英语成绩20%，计算每个学生加权总成绩，并按Map返回，key=学生姓名，value=学生加权后的总成绩。

```
/**
 * collect 流->集合
 * peek 处理流中的List数据
 * Collectors.groupingBy 分组
 * Collectors.summingDouble double类型数据求和
 * @param students 学生信息
 * @return key=学号，value=总成绩。
 */
private Map<Long, Double> calcWeightTotalScore(List<Student> students) {
    return students.stream()
```

```
            .peek(student -> {
                if (student.getCourse().equals("语文")) {
                    student.setScore(student.getScore() * 0.3);
                } else if (student.getCourse().equals("数学")) {
                    student.setScore(student.getScore() * 0.5);
                } else if (student.getCourse().equals("英语")) {
                    student.setScore(student.getScore() * 0.2);
                }
            })
            .collect(Collectors.groupingBy(Student::getStudentNumber,
Collectors.summingDouble(Student::getScore)));
}
```

com.coderbuff.chapter5_action.chapter5_1.Scene8

> 对于比较复杂的场景，我建议谨慎使用Stream。

第六章 调试与重构

6.1 调试

Stream让人心生胆怯的原因还有一个原因——调试。

传统的for循环，尽管逻辑写得复杂，但只需要打一个断点，总还能看得懂写的什么。但Stream对于部分人来讲，如果不了解Stream流，连断点可能都不知道在哪里打。

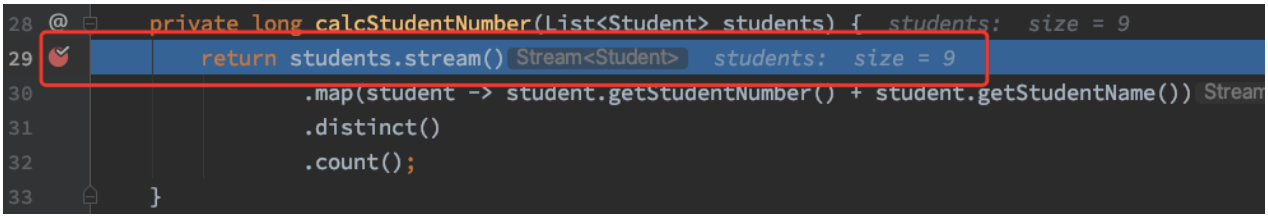
所以这也是部分人“痛批”Stream的原因，认为使用Stream都是在“炫技”。

实际上IDEA编译器从2019年的版本已经新增了对Stream流的调试功能。

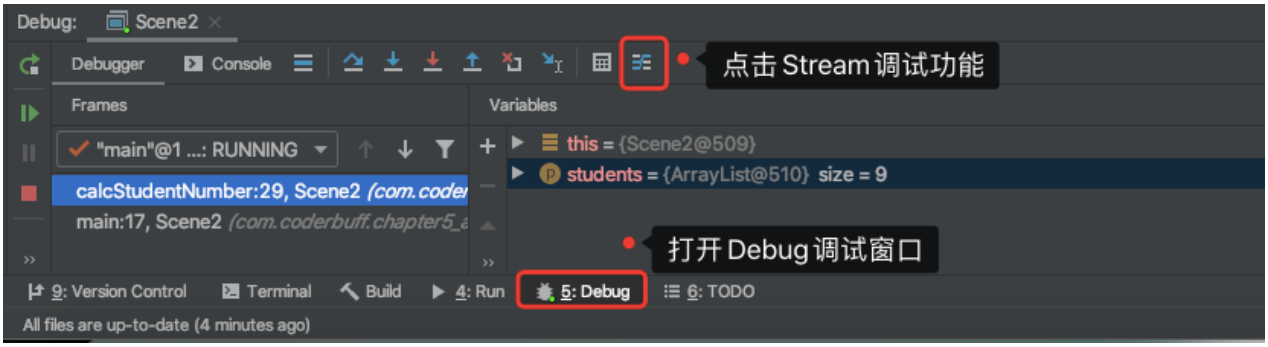
我们以“第五章 Stream流编码实战”的场景二为例，对它在IDEA中进行调试。

```
/**
 * map 按学生的学号+姓名转换为新的数据结构类型
 * distinct 去重
 * count 计数
 * @param students 学生
 * @return 去重后的学生数量
 */
private long calcStudentNumber(List<Student> students) {
    return students.stream()
            .map(student -> student.getStudentNumber() +
student.getStudentName())
            .distinct()
            .count();
}
```

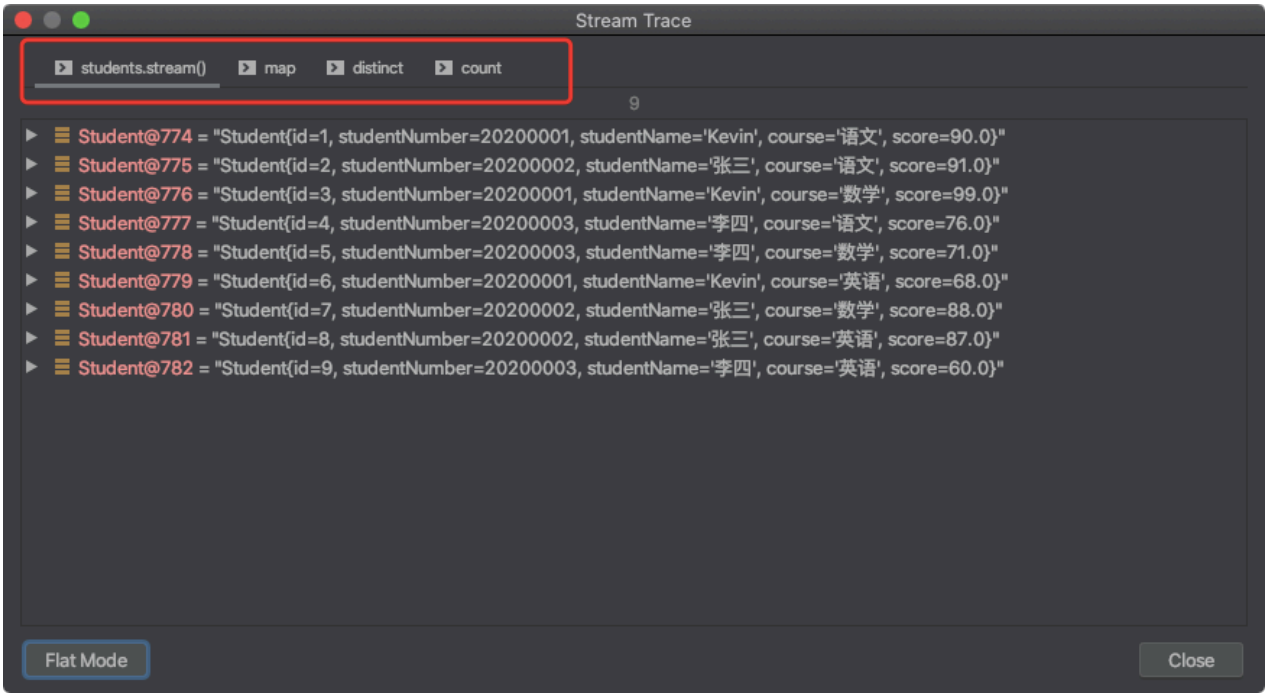
我们将断点打在调用`stream`这行，并以debug方式运行。



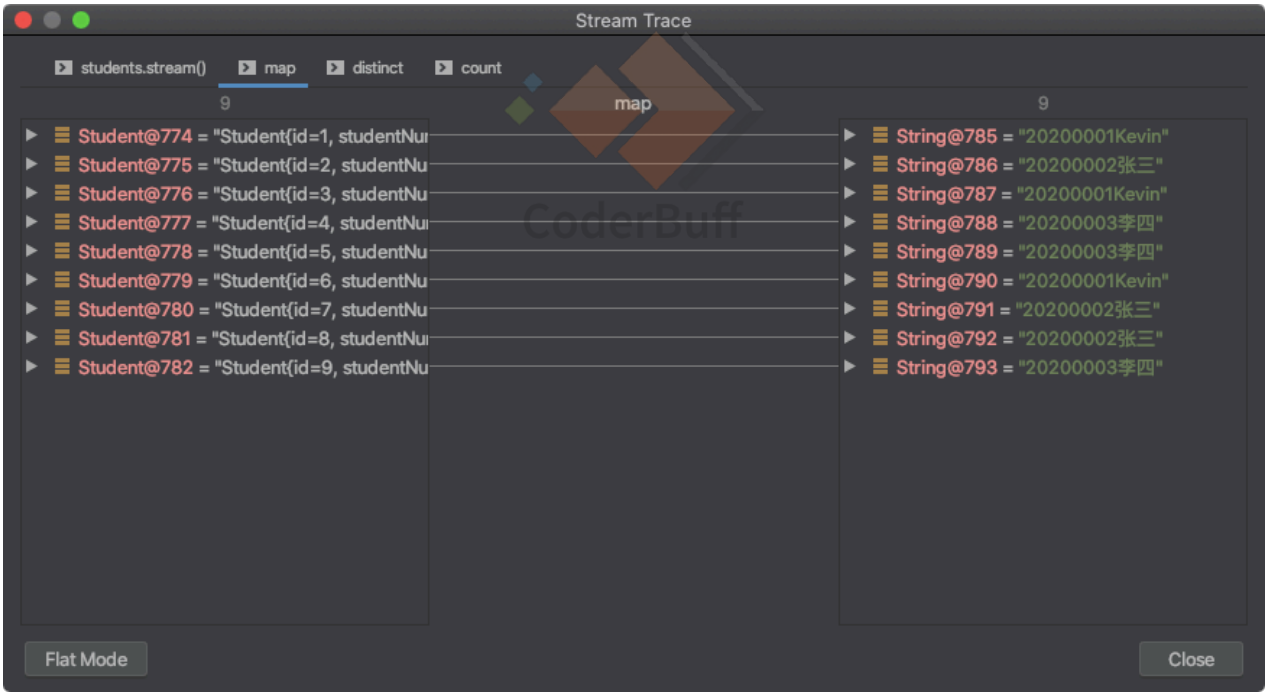
接下来程序在断点处暂停，打开Debug调试窗口，并点击Stream调试功能。



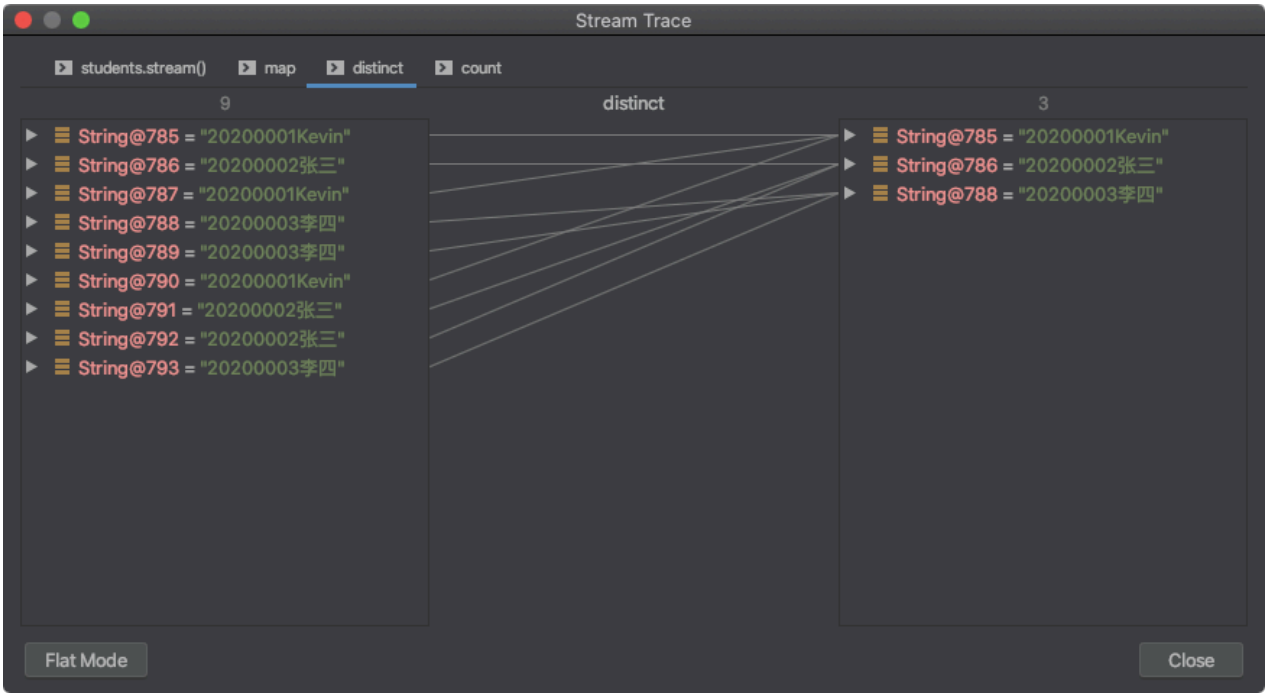
打开Stream调试功能后，看到的如下的窗口：



上方是每一个tab代表Stream流的每一个操作，点击每一个操作可以看到整个Stream的内部运行过程，例如`map`操作，我们把学号和姓名重新组合起来：



`distinct`去重操作：



也可以直接点击窗口下方“Flat Mode”总览整个Stream流的操作过程：

可以看到借助强大的IDEA，Stream调试也变得易如反掌。

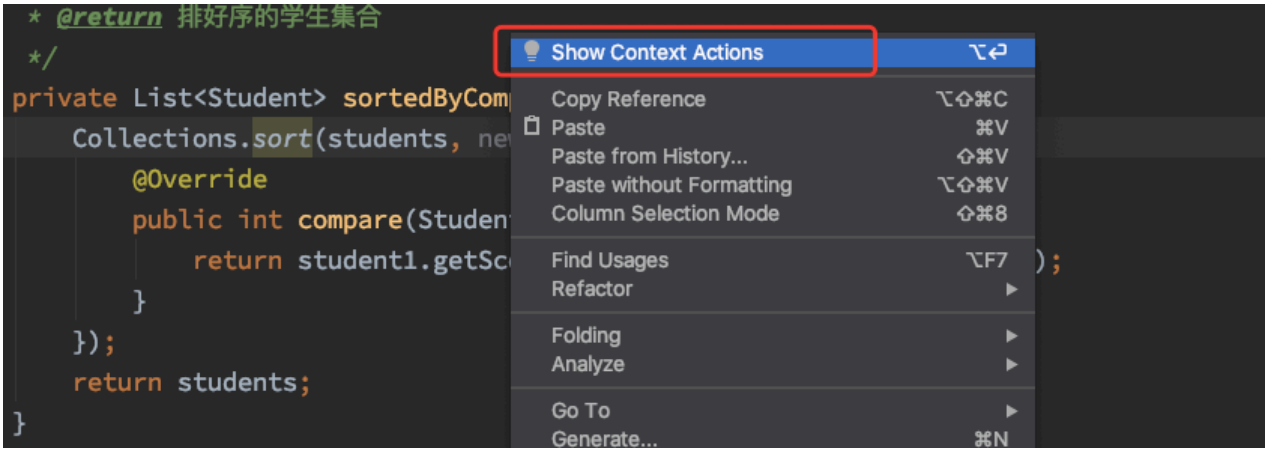
6.2 重构

我们同样可以借助IDEA对“老式”代码重构为更为现代的Lambda表达式。

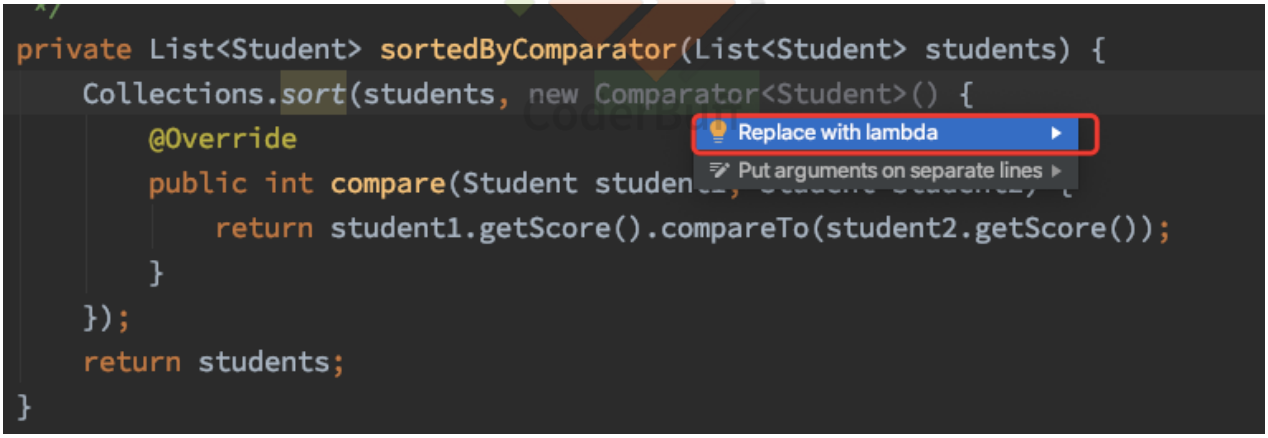
当你的代码中出现类似匿名内部类的可以优化的代码时，IDEA编译器会将它置灰显示：

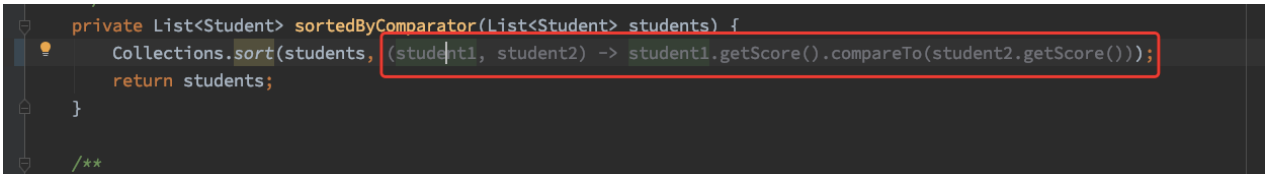
```
    private List<Student> sortedByComparator(List<Student> students) {
        Collections.sort(students, new Comparator<Student>() {
            @Override
            public int compare(Student student1, Student student2) {
                return student1.getScore().compareTo(student2.getScore());
            }
        });
        return students;
    }
```

我们将鼠标移到置灰处，单击鼠标右键选择“Show Context Actions”：



出现这样的提示语，点击“Replace with lambda”：





用Lambda表达式替换后，我们发现还有置灰，重复上面的操作后，最终被重构为最简洁的代码：

```
    private List<Student> sortedByComparator(List<Student> students) {
        Collections.sort(students, Comparator.comparing(Student::getScore));
        return students;
    }
```

这是编译器能够帮到我们的地方，仅仅限于语法上，如果有逻辑上的重构，仍然需要自己写好单元测试类反复测试进行重构。