Stream API 学会这样用，简化代码真牛批！

Java技术栈

A作者：何甜甜在吗

www.juejin.im/post/5d8226d4e51d453c135c5b9a

Java8的新特性主要是Lambda表达式和流，当流和Lambda表达式结合起来一起使用时，因为流申明式处理数据集合的特点，可以让代码变得简洁易读。

放大招，流如何简化代码

如果有一个需求，需要对数据库查询到的菜肴进行一个处理：

- 筛选出卡路里小于400的菜肴
- 对筛选出的菜肴进行一个排序
- 获取排序后菜肴的名字

菜肴：Dish.java

```
public class Dish {
    private String name;
    private boolean vegetarian;
    private int calories;
    private Type type;
```

```
    private Type type;

    // getter and setter
}
```

Java8以前的实现方式

```
private List<String> beforeJava7(List<Dish> dishList) {
        List<Dish> lowCaloricDishes = new ArrayList<>();

        //1.筛选出卡路里小于400的菜肴
        for (Dish dish : dishList) {
            if (dish.getCalories() < 400) {
                lowCaloricDishes.add(dish);
            }
        }

        //2.对筛选出的菜肴进行排序
        Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
            @Override
            public int compare(Dish o1, Dish o2) {
                return Integer.compare(o1.getCalories(), o2.getCalories());
            }
        });

        //3.获取排序后菜肴的名字
        List<String> lowCaloricDishesName = new ArrayList<>();
        for (Dish d : lowCaloricDishes) {
            lowCaloricDishesName.add(d.getName());
        }

        return lowCaloricDishesName;
    }
```

Java8之后的实现方式

```
private List<String> afterJava8(List<Dish> dishList) {
        return dishList.stream()
                .filter(d -> d.getCalories() < 400)  //筛选出卡路里小于400的菜肴
                .sorted(comparing(Dish::getCalories))  //根据卡路里进行排序
                .map(Dish::getName)  //提取菜肴名称
                .collect(Collectors.toList()); //转换为List
    }
```

不拖泥带水，一气呵成，原来需要写24代码实现的功能现在只需5行就可以完成了。JDK8的排序大法，这算也牛逼，看下

法，这篇也不例外，看下。

高高兴兴写完需求这时候又有新需求了，新需求如下：

- 对数据库查询到的菜肴根据菜肴种类进行分类，返回一个Map<Type, List<Dish>>的结果

这要是放在jdk8之前肯定会头皮发麻

Java8以前的实现方式

```
private static Map<Type, List<Dish>> beforeJdk8(List<Dish> dishList) {
    Map<Type, List<Dish>> result = new HashMap<>();

    for (Dish dish : dishList) {
        //不存在则初始化
        if (result.get(dish.getType())==null) {
            List<Dish> dishes = new ArrayList<>();
            dishes.add(dish);
            result.put(dish.getType(), dishes);
        } else {
            //存在则追加
            result.get(dish.getType()).add(dish);
        }
    }

    return result;
}
```

还好jdk8有**Stream**，再也不用担心复杂集合处理需求，关注微信公众号：Java技术栈，在后台回复：新特性，可以获取我整理的 N 篇最新新特性教程，都是干货。

Java8以后的实现方式

```
private static Map<Type, List<Dish>> afterJdk8(List<Dish> dishList) {
    return dishList.stream().collect(groupingBy(Dish::getType));
}
```

又是一行代码解决了需求，忍不住大喊**Stream** API牛批 看到流的强大功能了吧，接下来将详细介绍流

什么是流

流是从支持数据处理操作的源生成的元素序列，源可以是数组、文件、集合、函数。流不是集合元素，它不是数据结构并不保存数据，它的主要目的在于计算。

如何生成流

生成流的方式主要有五种

1.通过集合生成，应用中最常用的一种

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> stream = integerList.stream();
```

通过集合的stream方法生成流

2.通过数组生成

```
int[] intArr = new int[]{1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(intArr);
```

通过Arrays.stream方法生成流，并且该方法生成的流是数值流【即IntStream】而不是Stream<Integer>。补充一点使用数值流可以避免计算过程中拆箱装箱，提高性能。

Stream API提供了mapToInt、mapToDouble、mapToLong三种方式将对象流【即Stream】转换成对应的数值流，同时提供了boxed方法将数值流转换为对象流

3.通过值生成

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

通过Stream的of方法生成流，通过Stream的empty方法可以生成一个空流

4.通过文件生成

```
Stream<String> lines = Files.lines(Paths.get("data.txt"), Charset.defaultCharset())
```

通过Files.line方法得到一个流，并且得到的每个流是给定文件中的一行

5.通过函数生成 提供了iterate和generate两个静态方法从函数中生成流

iterator

```
Stream<Integer> stream = Stream.iterate(0, n -> n + 2).limit(5);
```

iterate方法接受两个参数，第一个为初始化值，第二个为进行的函数操作，因为iterator生成的流为无限流，通过limit方法对流进行了截断，只生成5个偶数

generator

```
Stream<Double> stream = Stream.generate(Math::random).limit(5);
```

generate方法接受一个参数，方法参数类型为Supplier，由它为流提供值。generate生成的流也是无限流，因此通过limit对流进行了截断

流的操作类型

流的操作类型主要分为两种

1.中间操作

一个流可以后面跟随零个或多个中间操作。其目的主要是打开流，做出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用。
这类操作都是惰性化的，仅仅调用到这类方法，并没有真正开始流的遍历，真正的遍历需等到终端操作时，常见的中间操作有下面即将介绍的filter、map等

2.终端操作

一个流有且只能有一个终端操作，当这个操作执行后，流就被关闭了，无法再被操作，因此一个流只能被遍历一次，若想在遍历需要通过源数据在生成流。终端操作的执行，才会真正开始流的遍历。如下面即将介绍的count、collect等

流使用

流的使用将分为终端操作和中间操作进行介绍

中间操作

filter筛选

```
List<Integer> integerList = Arrays.asList(1, 1, 2, 3, 4, 5);
Stream<Integer> stream = integerList.stream().filter(i -> i > 3);
```

通过使用filter方法进行条件筛选，filter的方法参数为一个条件

distinct去除重复元素

```
List<Integer> integerList = Arrays.asList(1, 1, 2, 3, 4, 5);
Stream<Integer> stream = integerList.stream().distinct();
```

通过distinct方法快速去除重复的元素

limit返回指定流个数

```
List<Integer> integerList = Arrays.asList(1, 1, 2, 3, 4, 5);
Stream<Integer> stream = integerList.stream().limit(3);
```

通过limit方法指定返回流的个数，limit的参数值必须>=0，否则将会抛出异常

skip跳过流中的元素

```
List<Integer> integerList = Arrays.asList(1, 1, 2, 3, 4, 5);
Stream<Integer> stream = integerList.stream().skip(2);
```

通过skip方法跳过流中的元素，上述例子跳过前两个元素，所以打印结果为2,3,4,5，skip的参数值必须>=0，否则将会抛出异常

map流映射

所谓流映射就是将接受的元素映射成另外一个元素

```
List<String> stringList = Arrays.asList("Java 8", "Lambdas", "In", "Action");
Stream<Integer> stream = stringList.stream().map(String::length);
```

通过map方法可以完成映射，该例子完成中String -> Integer的映射，之前上面的例子通过map方法完成了Dish->String的映射

flatMap流转换

将一个流中的每个值都转换为另一个流

```
List<String> wordList = Arrays.asList("Hello", "World");
```

```
List<String> strList = wordList.stream()
        .map(w -> w.split(" "))
        .flatMap(Arrays::stream)
        .distinct()
        .collect(Collectors.toList());
```

map(w -> w.split(" "))的返回值为Stream<String[]>，我们想获取Stream<String>，可以通过flatMap方法完成Stream ->Stream的转换。关注微信公众号：Java技术栈，在后台回复：新特性，可以获取我整理的 N 篇最新 Java 新特性教程，都是干货。

元素匹配

提供了三种匹配方式

1.allMatch匹配所有

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
if (integerList.stream().allMatch(i -> i > 3)) {
    System.out.println("值都大于3");
}
```

通过allMatch方法实现

2.anyMatch匹配其中一个

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
if (integerList.stream().anyMatch(i -> i > 3)) {
    System.out.println("存在大于3的值");
}
```

等同于

```
for (Integer i : integerList) {
    if (i > 3) {
        System.out.println("存在大于3的值");
        break;
    }
}
```

存在大于3的值则打印，java8中通过anyMatch方法实现这个功能

3.noneMatch全部不匹配

3.noneMatch全部不匹配

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
if (integerList.stream().noneMatch(i -> i > 3)) {
    System.out.println("值都小于3");
}
```

通过noneMatch方法实现

终端操作

统计流中元素个数

1.通过count

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
Long result = integerList.stream().count();
```

通过使用count方法统计出流中元素个数

2.通过counting

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
Long result = integerList.stream().collect(counting());
```

最后一种统计元素个数的方法在与collect联合使用的时候特别有用

查找

提供了两种查找方式

1.findFirst查找第一个

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> result = integerList.stream().filter(i -> i > 3).findFirst();
```

通过findFirst方法查找到第一个大于三的元素并打印

2.findAny随机查找一个

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
```

```
Optional<Integer> result = integerList.stream().filter(i -> i > 3).findAny();
```

通过findAny方法查找到其中一个大于三的元素并打印，因为内部进行优化的原因，当找到第一个满足大于三的元素时就结束，该方法结果和findFirst方法结果一样。

提供findAny方法是为了更好的利用并行流，findFirst方法在并行上限制更多【本篇文章将不介绍并行流】

reduce将流中的元素组合起来

假设我们对一个集合中的值进行求和

jdk8之前

```
int sum = 0;
for (int i : integerList) {
sum += i;
}
```

jdk8之后通过reduce进行处理

```
int sum = integerList.stream().reduce(0, (a, b) -> (a + b));
```

一行就可以完成，还可以使用方法引用简写成：

```
int sum = integerList.stream().reduce(0, Integer::sum);
```

reduce接受两个参数，一个初始值这里是0，一个BinaryOperator<T> accumulator
来将两个元素结合起来产生一个新值，

另外reduce方法还有一个没有初始化值的重载方法

获取流中最小最大值

通过min/max获取最小最大值

```
Optional<Integer> min = menu.stream().map(Dish::getCalories).min(Integer::compareTo);
Optional<Integer> max = menu.stream().map(Dish::getCalories).max(Integer::compareTo);
```

也可以写成：

也可以写成：

```
OptionalInt min = menu.stream().mapToInt(Dish::getCalories).min();
OptionalInt max = menu.stream().mapToInt(Dish::getCalories).max();
```

min获取流中最小值，max获取流中最大值，方法参数为Comparator<? super T> comparator

通过minBy/maxBy获取最小最大值

```
Optional<Integer> min = menu.stream().map(Dish::getCalories).collect(minBy(Integer::compareTo));
Optional<Integer> max = menu.stream().map(Dish::getCalories).collect(maxBy(Integer::compareTo));
```

minBy获取流中最小值，maxBy获取流中最大值，方法参数为Comparator<? super T> comparator

通过reduce获取最小最大值

```
Optional<Integer> min = menu.stream().map(Dish::getCalories).reduce(Integer::min);
Optional<Integer> max = menu.stream().map(Dish::getCalories).reduce(Integer::max);
```

求和

通过summingInt

```
int sum = menu.stream().collect(summingInt(Dish::getCalories));
```

如果数据类型为double、long，则通过summingDouble、summingLong方法进行求和

通过reduce

```
int sum = menu.stream().map(Dish::getCalories).reduce(0, Integer::sum);
```

通过sum

```
int sum = menu.stream().mapToInt(Dish::getCalories).sum();
```

在上面求和、求最大值、最小值的时候，对于相同操作有不同的方法可以选择执行。

可以选择collect、reduce、min/max/sum方法，推荐使用min、max、sum方法。因为它最简洁易读，同时通过mapToInt将对象流转换为数值流，避免了装箱和拆箱操作

通过averagingInt求平均值

```
double average = menu.stream().collect(averagingInt(Dish::getCalories));
```

如果数据类型为double、long，则通过averagingDouble、averagingLong方法进行求平均

通过summarizingInt同时求总和、平均值、最大值、最小值

```
IntSummaryStatistics intSummaryStatistics = menu.stream().collect(summarizingInt(Dish::getCalories));
double average = intSummaryStatistics.getAverage();  //获取平均值
int min = intSummaryStatistics.getMin();  //获取最小值
int max = intSummaryStatistics.getMax();  //获取最大值
long sum = intSummaryStatistics.getSum();  //获取总和
```

如果数据类型为double、long，则通过summarizingDouble、summarizingLong方法

通过foreach进行元素遍历

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
integerList.stream().forEach(System.out::println);
```

而在jdk8之前实现遍历：

```
for (int i : integerList) {
    System.out.println(i);
}
```

jdk8之后遍历元素来的更为方便，原来的for-each直接通过foreach方法就能实现了

返回集合

```
List<String> strings = menu.stream().map(Dish::getName).collect(toList());
Set<String> sets = menu.stream().map(Dish::getName).collect(toSet());
```

只举例了一部分，还有很多其他方法 jdk8之前

```
List<String> stringList = new ArrayList<>();
    Set<String> stringSet = new HashSet<>();
    for (Dish dish : menu) {
```

```
        stringList.add(dish.getName());
        stringSet.add(dish.getName());
}
```

通过遍历和返回集合的使用发现流只是把原来的外部迭代放到了内部进行，这也是流的主要特点之一。内部迭代可以减少好多代码量。

通过joining拼接流中的元素

```
String result = menu.stream().map(Dish::getName).collect(Collectors.joining(", "));
```

默认如果不通过map方法进行映射处理拼接的toString方法返回的字符串，joining的方法参数为元素的分界符，如果不指定生成的字符串将是一串的，可读性不强

进阶通过groupingBy进行分组

```
Map<Type, List<Dish>> result = dishList.stream().collect(groupingBy(Dish::getType));
```

在collect方法中传入groupingBy进行分组，其中groupingBy的方法参数为分类函数。还可以通过嵌套使用groupingBy进行多级分类

```
Map<Type, List<Dish>> result = menu.stream().collect(groupingBy(Dish::getType,
        groupingBy(dish -> {
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
        })));
```

进阶通过partitioningBy进行分区

分区是特殊的分组，它分类依据是true和false，所以返回的结果最多可以分为两组

```
Map<Boolean, List<Dish>> result = menu.stream().collect(partitioningBy(Dish :: isVegetarian))
```

等同于

```
Map<Boolean, List<Dish>> result = menu.stream().collect(groupingBy(Dish :: isVegetarian))
```

这个例子可能并不能看出分区和分类的区别，甚至觉得分区根本没有必要，换个明显一点的例子：

```
List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5);
Map<Boolean, List<Integer>> result = integerList.stream().collect(partitioningBy(i -> i < 3));
```

返回值的键仍然是布尔类型，但是它的分类是根据范围进行分类的，分区比较适合处理根据范围进行分类

总结

通过使用**Stream** API可以简化代码，同时提高了代码可读性，赶紧在项目里用起来。讲道理在没学**Stream** API之前，谁要是给我在应用里写很多**Lambda**，**Stream** API，飞起就想给他一脚。

我想，我现在可能爱上他了【嘻嘻】。同时使用的时候注意不要将声明式和命令式编程混合使用，前几天刷segment刷到一条：

用stream还能怎么优化下面代码？

java 125 次浏览

0

问题描述

用stream还能怎么优化下面代码？

问题出现的环境背景及自己尝试过哪些方法

相关代码

// 请把代码文本粘贴到下方（请勿用图片代替代码）

```
List<Integer> goodsIdList = new ArrayList<>();
refundOrderVO.getOrderItemList().forEach(refundOrderItem -> {
    orderItemList.forEach(orderItem -> {
        if (Objects.equals(refundOrderItem.getRefundOrderItemNo(),
             orderItem.getOrderItemNo())) {
            goodsIdList.add(orderItem.getGoodsId());
        }
 });
```

});

你期待的结果是什么？实际看到的错误信息又是什么？

更简的写法

imango老哥说的很对，别用声明式编程的语法干命令式编程的勾当

4

不知道具体场景，给一个中立写法

```
Set<String> refundOrderItemNos refundOrderItem.stream()
    .map(item -> item.getRefundOrderItemNo())
    .collect(Collectors.toSet());
```

```
List<Integer> goodsIdList = orderItemList.stream()
    .filter(item -> refundOrderItemNos.contains(item.getOrderItemNo))
    .map(item -> orderItem.getGoodsId())
    .collect(Collectors.toList());
```

评论 ¥赞赏 编辑 ···

TNT RP 2k
2019-09-11 更新

1

按照@_TNT_的写法应该是比较合适的，不过个人只是想提一点，用 `stream` 在处理集合的时候，如果你只是想着把以前的 `for` 循环改为 `stream` 里的 `forEach` ，那你还不如不用 `stream`

你得从思想上该改变哈，一个是命令式编程，一个是声明式编程（可能更进一步说是函数式编程）

简单来说，你用声明式编程的语法却干着命令式编程的"勾当" (⊙_⊙)

先在循环外面初始化一个 `List<Integer> goodsIdList` ，然后再在循环里找到需要的数据添加进去。。。这就是命令式编程石锤了，并且你使用 `stream` 看不到任何比较细力度的"函数"。。。全是一个大代码块。。就是两个大循环，你可以对比哈@_TNT_的写法，就看得出来区别了ヽ(ﾟ▽ﾟ)ﾉ

评论 ¥赞赏 编辑 ···

imango RP 1.3k
5 天前更新

---------- END ----------

学习资料：

分享一份最新 Java 架构师学习资料

100 本 Java 架构师重磅电子书！

最近热文：

1、**过年回家，反借钱攻略！**

2、20200202，千年难遇啊！

3、YYYY-MM-DD 的黑锅，我们不背！

4、京东把 Elasticsearch 用的真牛逼！

5、IDEA 公司推出新字体，极度舒适~

Java干货：

1、如何编写可怕的 Java 代码？

2、Oracle JDK 和 OpenJDK 有什么区别？

3、Java 14 令人期待的 5 大新特性！

4、Java中的对象都是在堆上分配的吗？

5、图文并茂，傻瓜都能看懂的JVM内存布局

Spring干货：

1、Spring 事务失效的 8 大原因，吊打面试官！

2、Spring 面试 7 大问题，你顶得住不？

3、Spring Boot 之配置导入，强大到不行！

4、Spring Cloud Greenwich最后计划版本发布！

5、Spring Cloud 升级最新 Greenwich 版本

本公众号干货实在太多了，没法都搬上来，扫码关注**Java技术栈**公众号，获取更多最主流的 Java 技术干货。

点击「阅读原文」带你飞~

阅读原文