原 java 8（二）--函数式数据处理Stream

2017年07月11日 14:24:39

一、前言

有一个池塘，池塘里面有各种各样的鱼虾，假如我想找到所有大于2斤小于3斤的草鱼的名字（假如鱼有名字）。

这个需求很简单，映射到数据库中可以这样查询：

SELECT name FROM tbl_fishes WHERE kind = 'grass carp' AND weight < 3 AND WHERE weight > 2

这条SQL语句简单明了，剩下的就交给数据库搞定了，那用java如何实现呢?

给定一个集合，一遍一遍的迭代，似乎别无他法，而且还要考虑并行，实在是头疼，但是java 8流处理出来后，问题就简单多了。

```
fishes.stream()
.filter(f -> f.getWeight < 3)
.filter(f -> f.getWeight > 2)
.filter(f -> f.getKind().equals("grass carp")
.map(f -> f.getName())
```

而且还可以并行处理。

二、流简介

2.1、什么是流

流到底是什么呢？简短的定义就是“**从源生成的支持数据处理操作的元素序列**”。

- 元素序列——就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。因为集合是数据结构，所以它的主要目的是以特定的
  杂度存储和访问元素（如ArrayList 与 LinkedList）。但流的目的在于表达计算，比如、filter、sorted和map。集合讲的是数据，流讲的是计算。
- 源——流会使用一个提供数据的源，如集合、数组或输入/输出资源。 请注意，从有序集合生成流时会保留原有的顺序。由列表生成的流，其元
  一致。
- 数据处理操作——流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match、s
  可以顺序执行，也可并行执行。

上java 8实战中的代码说明：

```
public class Test1
{
public static void main(String[] args) {
List<Dish> menu = Arrays.asList(
new Dish("pork", false, 800, Dish.Type.MEAT),
new Dish("beef", false, 700, Dish.Type.MEAT),
new Dish("chicken", false, 400, Dish.Type.MEAT),
new Dish("french fries", true, 530, Dish.Type.OTHER),
new Dish("rice", true, 350, Dish.Type.OTHER),
new Dish("season fruit", true, 120, Dish.Type.OTHER),
new Dish("pizza", true, 550, Dish.Type.OTHER),
new Dish("prawns", false, 300, Dish.Type.FISH),
new Dish("salmon", false, 450, Dish.Type.FISH) );

List<String> threeHighCaloricDishNames =
menu.stream()
.filter(d -> d.getCalories() > 300)
.map(Dish::getName)
```

```
.limit(3)
.collect(toList());
System.out.println(threeHighCaloricDishNames);
}
}

class Dish {
private final String name;
private final boolean vegetarian;
private final int calories;
private final Type type;
public Dish(String name, boolean vegetarian, int calories, Type type) {
this.name = name;
this.vegetarian = vegetarian;
this.calories = calories;
this.type = type;
}
public String getName() {
return name;
}
public boolean isVegetarian() {
return vegetarian;
}
public int getCalories() {
return calories;
}
public Type getType() {
return type;
}
@Override
public String toString() {
return name;
}
public enum Type { MEAT, FISH, OTHER }
}
```

数据**源**是菜肴列表，它给流提供一个**元素序列**。接下来，对流应用一系列**数据处理操作**：filter、map、limit和collect。

2.2、流的好处

- 声明性——更简洁，更易读
- 可复合——更灵活
- 可并行——性能更好

> PS：
>
> 参考：http://www.importnew.com/24250.html
>
> 对于简单操作，比如最简单的遍历，Stream串行API性能明显差于显示迭代，但并行的Stream API能够发挥多核特性。
>
> 对于复杂操作，Stream串行API性能可以和手动实现的效果匹敌，在并行执行时Stream API效果远超手动实现。

2.3、流与集合的区别

粗略地说，集合与流之间的差异就在于什么时候进行计算。

- 集合是一个**内存中的数据结构，它包含数据结构中目前所有的值**——集合中的每个元素都得先算出来才能添加到集合中。可以往集合里加东西但是不管什么时候，集合中的每个元素都是放在内存里的，元素都得先算出来才能成为集合的一部分。
- 流则是在概念上固定的数据结构（不能添加或删除元素），其元素则是按需计算的。

举个例子：

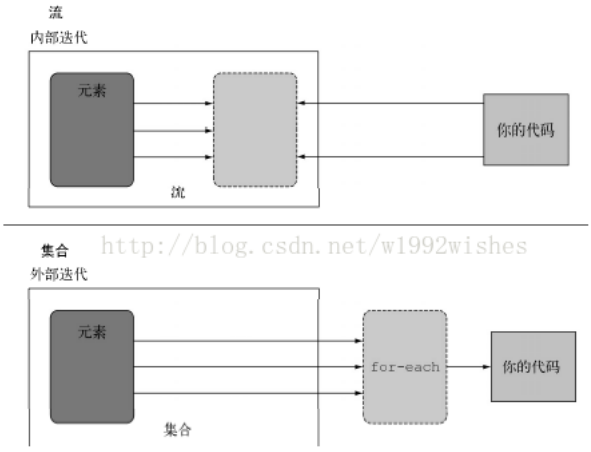
我的电脑E盘有一个音乐的文件夹，里面下载了很多歌曲，那么这个文件夹就像是java里的集合，歌曲就像集合中的元素，我想要听哪首歌曲，戴上耳机就可以享受音乐里面的歌曲都下载好了，就像前面说的元素得先计算出来。

我一直很喜欢赵雷，老早就听说他出了新专辑--《无法长大》，可是我的文件夹下面又没有这些歌曲，怎么办呢？第一个方法当然是下载下来，存放到文件夹下面，就元素；另一种呢，我可以选择在线收听，我不必下载，网易云会自动帮我缓存好我将要播放的部分，这时同样可以欣赏赵雷忧郁的歌声。

不必下载，缓存将要听的部分就好比流，是按需计算的。

2.4、流的特点

- 只能遍历一次：和迭代器类似，流只能遍历一次。遍历完之后，这个流已经被消费掉了，再次遍历，就会抛出异常。
- 内部迭代：使用Collection接口需要用户去做迭代（比如用for-each），这称为外部迭代。 相反，
  Streams库使用内部迭代——它帮把迭代做了，还把得到的流值存在了某个地方，只要给出一个函数说要干什么就可以了。



使用for-each需要明确指定先做什么，再做什么，比如去菜地里摘菜，先摘韭菜，再摘豆角，然后摘西红柿，但内部迭代不一样，只要声明去摘菜，那么可以
柿，再摘中间的豆角，最后摘韭菜，也可以先摘一些韭菜，再去摘西红柿，然后又回过来摘韭菜。

内部迭代时，项目可以透明地并行处理，或者用更优化的顺序进行处理。Streams库的内部迭代可以自动选择一种适合硬件的数据表示和并行实现。

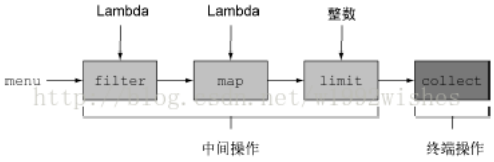
2.5、流的操作

java.util.stream.Stream中的Stream接口定义了许多操作。它们可以分为两大类。

```
menu.stream()
.filter(d -> d.getCalories() > 300)
.map(Dish::getName)
.limit(3)
.collect(toList());
```

- filter、map和limit可以连成一条流水线；
- collect触发流水线执行并关闭它。

可以连接起来的流操作称为**中间操作**，关闭流的操作称为**终端操作**。



2.5.1、中间操作

诸如filter或sorted等中间操作会返回另一个流。这让多个操作可以连接起来形成一个查询。重要的是，除非流水线上触发一个终端操作，否则中间操作不会执行任何
懒。这是因为中间操作一般都可以合并起来，在终端操作时一次性全部处理。

2.5.2、终端操作

终端操作会从流的流水线生成结果。

三、使用流

空谈许久，现在上路吧，来看看Stream API支持的方便的操作。

3.1、筛选和切片

3.1.1、用Predicate筛选

Streams接口支持filter方法，该操作会接受一个Predicate作为参数，并返回一个包括所有符合谓词的元素的流。

例如，筛选出所有素菜，创建一张素食菜单：

```
List<Dish> vegetarianMenu = menu.stream()
.filter(Dish::isVegetarian)
.collect(toList());
```

3.1.2、distinct

流还支持一个叫作distinct的方法，它会返回一个元素各异（根据流所生成元素的hashCode和equals方法实现）的流。发现没，同SQL中的distinct一样。

3.1.3、截断流

流支持limit(n)方法，该方法会返回一个不超过给定长度的流。所需的长度作为参数传递给limit。如果流是有序的，则最多会返回前n个元素。

3.1.4、跳过流

流还支持skip(n)方法，返回一个扔掉了前n个元素的流。如果流中元素不足n个，则返回一个空流。limit(n)和skip(n)是互补的！

3.2、映射

一个非常常见的数据处理套路就是从某些对象中选择信息。比如在SQL里，你可以从表中选择一列。Stream API也通过map和flatMap方法提供了类似的工具。

3.2.1、map

流支持map方法，它会接受一个Function作为参数。这个Function会被应用到每个元素上，并将其映射成一个新的元素。

例如，把方法引用Dish::getName传给了map方法，来提取流中菜肴的名称：

```
List<String> dishNames = menu.stream()
.map(Dish::getName)
.collect(toList());
```

因为getName方法返回一个String，所以map方法输出的流的类型就是Stream<String>。

3.2.2、flatmap

map可以把Stream中的元素按照给定的Function进行转换，新生成的Stream只包含转换生成的元素。但如果Stream中的元素是集合，则无能为力，这个时候就需要flatn

flatMap：和map类似，不同的是其每个元素转换得到的是Stream对象，会把子Stream中的元素压缩到父集合中，就好像多个流都被合并起来，扁平化为一个流。

```
List<List<Integer>> outer = new ArrayList<>();
List<Integer> inner1 = new ArrayList<>();
inner1.add(1);
List<Integer> inner2 = new ArrayList<>();
inner2.add(2);
List<Integer> inner3 = new ArrayList<>();
inner3.add(3);
List<Integer> inner4 = new ArrayList<>();
inner4.add(4);
List<Integer> inner5 = new ArrayList<>();
inner5.add(5);
outer.add(inner1);
outer.add(inner2);
outer.add(inner3);
outer.add(inner4);
outer.add(inner5);
List<Integer> result = outer.stream().flatMap(inner -> inner.stream().map(i -> i + 1)).collect(toList());
System.out.println(result);
```

3.3、查找和匹配

3.3.1、anyMatch

anyMatch方法接收一个Predicate，用以匹配流中是否至少有一个元素符合Predicate。

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

anyMatch方法返回一个boolean，因此是一个终端操作。

3.3.2、allMatch

allMatch方法接收一个Predicate，用以匹配流中是否所有素都符合Predicate。

```
boolean isHealthy = menu.stream()
.allMatch(d -> d.getCalories() < 1000);
```

allMatch也是一个终端操作。

3.3.3、noneMatch

和allMatch相反

```
boolean isHealthy = menu.stream()
.noneMatch(d -> d.getCalories() >= 1000);
```

anyMatch、allMatch和noneMatch这三个操作都用到了短路，这就是Java中&&和||运算符短路在流中的版本。

> 短路求值
>
> 有些操作不需要处理整个流就能得到结果。例如，假设需要对一个用and连起来的大布 尔表达式求值。不管表达式有多长，只需找到一个表达式为false，就可以推
> 将返回false，所以用不着计算整个表达式。这就是短路。 对于流而言，某些操作（例如allMatch、anyMatch、noneMatch、findFirst和findAny） 不用处理整个
> 果。只要找到一个元素，就可以有结果了。同样，limit也是一个 短路操作：它只需要创建一个给定大小的流，而用不着处理流中所有的元素。在碰到无限大小 的流
> 操作就有用了：它们可以把无限流变成有限流。

3.3.4、findAny

findAny方法将返回当前流中的任意元素。它可以与其他流操作结合使用。

```
Optional<Dish> dish =
menu.stream()
.filter(Dish::isVegetarian)
.findAny();
```

3.3.5、findFirst

```
menu.stream()
.filter(Dish::isVegetarian)
.findAny()
.ifPresent(d -> System.out.println(d.getName());
```

3.4、归约

reduce操作可以用来进行一些复杂的查询，比如“计算菜单中的总卡路里”或“菜单中卡路里最高的菜是哪一个”。此类查询需要将流中所有元素反复结合起来，得到一个值
ger。这样的查询可以被归类为归约操作（将流归约成一个值）。

3.4.1、求和

java 8之前求和：

```
int sum = 0;
for (int x : numbers) {
sum += x;
}
```

java 8:

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

在Java 8中，Integer类现在有了一个静态的sum方法来对两个数求和，所以可以用方法引用表达：

```
int sum = numbers.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>来将两个元素结合起来产生一个新值，这里用的是lambda (a, b) -> a + b。

这里Lambda反复结合每个元素，直到流被归约成一个值。

3.4.2、最大值和最小值

reduce接受两个参数：

- 一个初始值
- 一个Lambda来把两个流元素结合起来并产生一个新值

所以也可以用来求最大值和最小值。

java 8中Integer中也有max和min方法：

```
Optional<Integer> max = numbers.stream().reduce(Integer::max);

Optional<Integer> min = numbers.stream().reduce(Integer::min);
```

四、简单实践

现有Customer和Orders两个类，代码如下：

```
class Customer{
private String name;
private int age;

public Customer(String name, int age) {
this.name = name;
this.age = age;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}
```

```
}

class Order{
private Customer customer;
private int year;
private int value;

public Order(Customer customer, int year, int value) {
this.customer = customer;
this.year = year;
this.value = value;
}

public Customer getCustomer() {
return customer;
}

public void setCustomer(Customer customer) {
this.customer = customer;
}

public int getYear() {
return year;
}

public void setYear(int year) {
this.year = year;
}

public int getValue() {
return value;
}

public void setValue(int value) {
this.value = value;
}
}
```

问题如下：

4.1、查询2014年后的订单，并按交易额排序。

```
public class Test2 {
public static void main(String[] args) {
Customer customer1 = new Customer("wawa",8);
Customer customer2 = new Customer("xiaoli", 11);
Customer customer3 = new Customer("dahuang", 22);
Customer customer4 = new Customer("cangbai", 14);
Customer customer5 = new Customer("cainiao", 25);

List<Order> orders = Arrays.asList(
new Order(customer1, 2015, 3000),
new Order(customer2, 22017, 4000),
new Order(customer3, 2011, 111),
new Order(customer4, 2014, 1000),
new Order(customer5, 2016, 33333)
);

orders.stream()
.filter(order -> order.getYear() >= 2014)
.sorted(Comparator.comparing(Order::getValue))
.collect(toList());
}
}
```

4.2、返回所有顾客的姓名字符串，按字母顺序排序

```
String results = orders.stream()
.map(order -> order.getCustomer().getName())
.distinct()
.sorted()
.reduce("", (n1, n2) -> n1 + n2);
```

4.3、有没有订单金额大于10000

```
boolean results = orders.stream().anyMatch( order -> order.getValue() > 10000);
```

4.4、打印所有顾客姓名

```
orders.stream().forEach(order -> System.out.println(order.getCustomer().getName()));
```

五、数值流

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

这段代码有一个问题，它有一个暗含的装箱成本。每个Integer都必须拆箱成一个原始类型，再进行求和。

Java 8引入了三个原始类型特化流接口来解决这个问题：IntStream、DoubleStream和LongStream，分别将流中的元素特化为int、long和double，从而避免了暗含的装

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

mapToInt会从每道菜中提取热量（用一个Integer表示），并返回一个IntStream。

数值流也可以转换为对象流：

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> stream = intStream.boxed();
```

补充：IntStream和LongStream有range和rangeClosed两个方法，可以生产一段范围内的数值流，其中range方法不包含结尾值，而rangeClosed包含结尾值。

六、构建流

6.1、由值创建流

静态方法Stream.of，通过显式值创建一个流。它可以接受任意数量的参数。

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

可以使用empty得到一个空流。

```
Stream<String> emptyStream = Stream.empty();
```

6.2、由数组创建流

静态方法Arrays.stream从数组创建一个流，它接受一个数组作为参数。

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();//是一个IntStream
```

6.3、由文件生成流

java.nio.file.Files中的很多静态方法都会返回一个流。例如，Files.lines，它会返回一个由指定文件中的各行构成的字符串流。

```
long uniqueWords = 0;
try(Stream<String> lines = Files.lines(Paths.get("data.txt"), Charset.defaultCharset())){
uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))
.distinct()
.count();
}
catch(IOException e){
}
```

使用Files.lines得到一个流，其中的每个元素都是给定文件中的一行line，对line调用split方法将行拆分成单词。最后，把distinct和count方法链接起来，数数流中有多少个不同的词。

6.4、由函数生成流

Stream API提供了两个静态方法来从函数生成流：Stream.iterate和Stream.generate。

这两个操作可以创建所谓的无限流：不像从固定集合创建的流那样有固定大小的流，它们创建的流会用给定的函数按需创建值，因此可以无穷无尽地计算下去！

6.4.1、iterate

```
Stream.iterate(0, n -> n + 2)
.limit(10)
.forEach(System.out::println);
```

iterate方法接受一个初始值，还有一个依次应用在每个产生的新值上的Lambda（UnaryOperator<t>类型）。

用Lambda n -> n + 2，返回的是前一个元素加上2。

6.4.2、generate

```
Stream.generate(Math::random)
.limit(5)
.forEach(System.out::println);
```

generate不是依次对每个新生成的值应用函数的。它接受一个Supplier<T>类型的Lambda提供新的值。

七、用流收集数据

java 8 java.util.stream包下Collectors类提供了很多工厂方法（例如toList）用来创建Collectors。这些方法主要提供了三大功能：

- 将流元素归约和汇总为一个值
- 元素分组
- 元素分区

7.1、归约和汇总

7.1.1、计算流中共有多少元素counting

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

7.1.2、查找流中的最大值和最小值maxBy/minBy

```
Comparator<Dish> dishCaloriesComparator = Comparator.comparingInt(Dish::getCalories);
Optional<Dish> mostCalorieDish = menu.stream().collect(maxBy(dishCaloriesComparator));
```

因为menu可能为空，所以返回Optional<Dish>。Optional下章再作介绍。

7.1.3、汇总summingInt

```
//菜单列表的总热量
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));

//菜单列表的平均热量
double avgCalories = menu.stream().collect(averagingInt(Dish::getCalories));
```

7.1.4、连接字符串joining

joining工厂方法返回的收集器会把对流中每一个对象应用toString方法得到的所有字符串连接成一个字符串：

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

但这样的字符串可读性不好，joining工厂方法有一个重载版本可以接受元素之间的分界符：

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

7.1.5、广义的归约汇总

前面的几种方法都可以看出reducing工厂方法定义的归约过程的特殊情况而已，Collectors.reducing工厂方法是所有这些特殊情况的一般化。比如，可以用reducing方法创建的收集器来计算菜单的总热量：

```
int totalCalories = menu.stream().collect(reducing(0, Dish::getCalories, (i, j) -> i + j));
```

reducing需要三个参数。

- 第一个参数是归约操作的**起始值**，也是流中没有元素时的返回值，所以很显然对于数值 和而言0是一个合适的值。
- 第二个参数是一个**转换函数**，将菜肴转换成一个表示其所含热量的int。
- 第三个参数是一个**BinaryOperator**，将两个项目累积成一个同类型的值。这里它就是 对两个int求和。

用reducing来找到热量最高的菜：

```
Optional<Dish> mostCalorieDish =
menu.stream().collect(reducing((d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2));
```

可以把单参数reducing工厂方法创建的收集器看作三参数方法的特殊情况，第一个项目作为起点，把恒等函数（即一个函数仅仅是返回其输入参数）作为一个转换函数。

7.2、分组

java 8也提供了类似数据库的分组操作。

现假设把菜单中的菜按照类型进行分类，用Collectors.groupingBy工厂方法返回的收集器就可以轻松地完成这项任务：

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

再假设想把热量不到400卡路里的菜划分为“低热量”（diet），热量400到700卡路里的菜划为“普通”（normal），高于700卡路里的划为“高热量”（fat）。因为没有Dish可用，不能用方法引用，可以用Lambda表达式：

```
public enum CaloricLevel { DIET, NORMAL, FAT }

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

还可以进行多级分组：

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
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;
} )
)
);
```

这在java 8之前，估计就只能循套循环才能实现，代码绝没有现在这么明白清晰。

groupingBy的第二个参数不只是可以用groupingBy，还可以是别的Collectors：

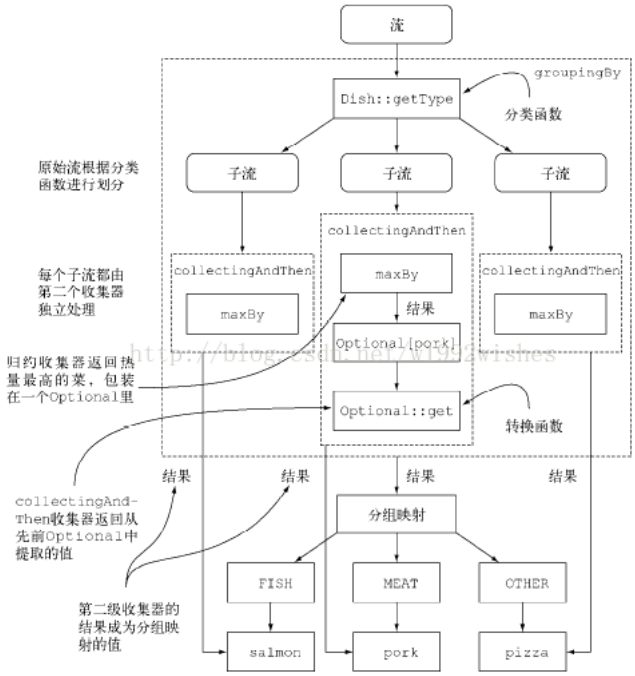
```
// 每种菜品有多少个
Map<Dish.Type, Long> typesCount = menu.stream().collect(groupingBy(Dish::getType, counting()));

// 每种菜品卡路里最高的菜，结果是一个map，以Dish的类型作为键，Optional<Dish>作为值
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
menu.stream().collect(
groupingBy(Dish::getType,
maxBy(comparingInt(Dish::getCalories))));
```

可以把收集器返回的结果转换为另一种类型，这里Optional::get操作放在这里是安全的，因为reducing收集器永远都不会返回Optional.empty()。groupingBy收集器只有后，第一次在流中找到某个键对应的元素时才会把键加入分组Map中。

```
Map<Dish.Type, Dish> mostCaloricByType =
menu.stream()
.collect(groupingBy(Dish::getType,
collectingAndThen(
maxBy(comparingInt(Dish::getCalories)),
Optional::get)));
```

其工作流程大概如下：

- 收集器用虚线表示，groupingBy是最外层，根据菜肴的类型把菜单流分组，得到三 个子流。
- groupingBy收集器包裹着collectingAndThen收集器，因此分组操作得到的每个子流 都用这第二个收集器做进一步归约。
- collectingAndThen收集器又包裹着第三个收集器maxBy。
- 随后由归约收集器进行子流的归约操作，然后包含它的collectingAndThen收集器会对 其结果应用Optional:get转换函数。
- 对三个子流分别执行这一过程并转换而得到的三个值，也就是各个类型中热量最高的 Dish，将成为groupingBy收集器返回的Map中与各个分类型）相关联的值。



7.3、分区

分区是分组的特殊情况：由一个谓词（返回一个布尔值的函数）作为分类函数。

把菜单按照素食和非素食分开：

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

找到素食和非素食中热量最高的菜：

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian =
menu.stream().collect(
partitioningBy(Dish::isVegetarian,
collectingAndThen(
maxBy(comparingInt(Dish::getCalories)),
Optional::get)));
```

八、并行流

可以用parallel方法将串行流转为并行流：

```
public static long parallelSum(long n) {
return Stream.iterate(1L, i -> i + 1)
.limit(n)
.parallel() //parallel将串行流变为并行流
.reduce(0L, Long::sum);
}
```

也可以用sequential将并行流转为并行流：

```
stream.parallel()
.filter(...)
.sequential()
.map(...)
.parallel()
.reduce();
```

但并不是把这两个方法结合起来，就可以更细化地控制在遍历流时哪些操作要并行执行，哪些要串行， 一次调用会影响整个流水线。

8.1、测试一下性能

```
public class ParallelStreams {
public static long measureSumPerf(Function<Long, Long> adder, long n) {
long fastest = Long.MAX_VALUE;
for (int i = 0; i < 10; i++) {
long start = System.nanoTime();
long sum = adder.apply(n);
long duration = (System.nanoTime() - start) / 1_000_000;
// System.out.println("Result: " + sum);
if (duration < fastest) fastest = duration;
}
return fastest;
}

//串行流
public static long sequentialSum(long n) {
return Stream.iterate(1L, i -> i + 1)
.limit(n)
}

// 传统for循环
public static long iterativeSum(long n) {
long result = 0;
for (long i = 1L; i <= n; i++) {
result += i;
}
return result;
}

//并行流
public static long parallelSum(long n) {
return Stream.iterate(1L, i -> i + 1)
.limit(n)
.parallel()
.reduce(0L, Long::sum);
}

public static void main(String[] args) {
System.out.println(Runtime.getRuntime().availableProcessors());
System.out.println("Sequential sum done in:" +
measureSumPerf(ParallelStreams::sequentialSum, 10_000_000) + " msecs");
System.out.println("Iterative sum done in:" +
measureSumPerf(ParallelStreams::iterativeSum, 10_000_000) + " msecs");
System.out.println("Parallel sum done in: " +
measureSumPerf(ParallelStreams::parallelSum, 10_000_000) + " msecs" );
}
}
```

结果发现，并行流性能最差，串行其次，而传统的for-each性能远远高于流处理：

> PS：这是对基本类型的测试，其实测试一下一个复杂一点的对象，串行流效率不比for-each差，甚至还有超越，而并行流效率更高，且随着核数越多，效率越高。并行流效率很差。

```
4
Sequential sum done in:114 msecs
Iterative sum done in:2 msecs
Parallel sum done in: 168 msecs
```

这里实际上有两个问题：

- iterate生成的是装箱的对象，必须拆箱成数字才能求和；
- 很难把iterate分成多个独立块来并行执行。（iterate很难分割成能够独立执行的小块，因为每次应用这个函数都要依赖前一次应用的结果。）

整张数字列表在归纳过程开始时没有准备好，因而无法有效地把流划分为小块来并行处理。把流标记成并行，其实是给顺序处理增加了开销，它还要把每次求和操作分程上。

这说明了并行编程可能很复杂，有时候甚至有点违反直觉。如果用得不对（比如采用了一个不易并行化的操作，如iterate），它甚至可能让程序的整体性能更差。

8.2、使用更有针对性的方法重新测试

```
public class ParallelStreams {
public static long measureSumPerf(Function<Long, Long> adder, long n) {
long fastest = Long.MAX_VALUE;
for (int i = 0; i < 10; i++) {
long start = System.nanoTime();
long sum = adder.apply(n);
long duration = (System.nanoTime() - start) / 1_000_000;
// System.out.println("Result: " + sum);
if (duration < fastest) fastest = duration;
}
return fastest;
}

//串行流
public static long sequentialSum(long n) {
return Stream.iterate(1L, i -> i + 1)
.limit(n)
.reduce(0L, Long::sum);
}

// 传统for循环
public static long iterativeSum(long n) {
long result = 0;
for (long i = 1L; i <= n; i++) {
result += i;
}
return result;
}

//并行流
public static long parallelSum(long n) {
return Stream.iterate(1L, i -> i + 1)
.limit(n)
.parallel()
.reduce(0L, Long::sum);
}

//使用更有针对性的方法 串行流
public static long rangedSum(long n) {
return LongStream.rangeClosed(1, n)
.reduce(0L, Long::sum);
}

//使用更有针对性的方法 并行流
public static long parallelRangedSum(long n) {
return LongStream.rangeClosed(1, n)
.parallel()
.reduce(0L, Long::sum);
}

public static void main(String[] args) {
System.out.println(Runtime.getRuntime().availableProcessors());
System.out.println("Sequential sum done in:" +
measureSumPerf(ParallelStreams::sequentialSum, 10_000_000) + " msecs");
System.out.println("Iterative sum done in:" +
measureSumPerf(ParallelStreams::iterativeSum, 10_000_000) + " msecs");
System.out.println("Parallel sum done in: " +
measureSumPerf(ParallelStreams::parallelSum, 10_000_000) + " msecs" );
System.out.println("Sequential ranged sum done in: " +
measureSumPerf(ParallelStreams::rangedSum, 10_000_000) + " msecs" );
System.out.println("Parallel ranged sum done in: " +
measureSumPerf(ParallelStreams::parallelRangedSum, 10_000_000) + " msecs" );
```

```
}
}
```

结果是：

```
4
Sequential sum done in:114 msecs
Iterative sum done in:2 msecs
Parallel sum done in: 164 msecs
Sequential ranged sum done in: 4 msecs
Parallel ranged sum done in: 1 msecs
```

串行流与for-each差距已经不大，而并行流效率则超过了for-each，是因为rangeClosed与iterate相比有两个优点：

- LongStream.rangeClosed直接产生原始类型的long数字，没有装箱拆箱的开销。
- LongStream.rangeClosed会生成数字范围，很容易拆分为独立的小块。

> PS：并行化是有代价的。并行化过程本身需要对流做递归划分，把每个子流的归纳操作分配到不同的线程，然后把这些操作的结果合并成一个值。但在多个
> 代价也可能很大，所以很重要的一点是要保证在内核中并行执行工作的时间比在内核之间传输数据的时间长。
>
> 同时也要注意，**并行流也应该避免共享可变状态。**

8.3、高效使用并行流的建议

- 最靠谱的，测试，如果存在疑问，就测试。
- 尽量使用IntStream、LongStream、DoubleStream这些流避免装箱。
- 有些操作本身在并行流上的性能就比顺序流差。特别是limit和findFirst等依赖于元素顺序的操作，它们在并行流上执行的代价非常大。
- 对于较小的数据量，选择并行流几乎从来都不是一个好的决定。
- 要考虑流背后的数据结构是否易于分解。例如，ArrayList的拆分效率比LinkedList高得多，因为前者用不着遍历就可以平均拆分，而后者则必须遍历。
- 还要考虑终端操作中合并步骤的代价是大是小。

附上一张“流的数据源和可分解性”相关表：

源	可分解性
ArrayList	极佳
LinkedList	差
IntStream.range	极佳
Sream.iterate	差
HashSet	好
TreeSet	好

文章标签： java 1.8 函数式 ▼查看关于本篇文章更多信息

上一篇 java 8（一）--Lambda表达式

下一篇 java 8（三）--用Optional取代null