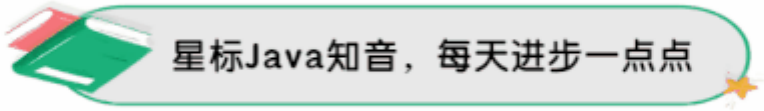
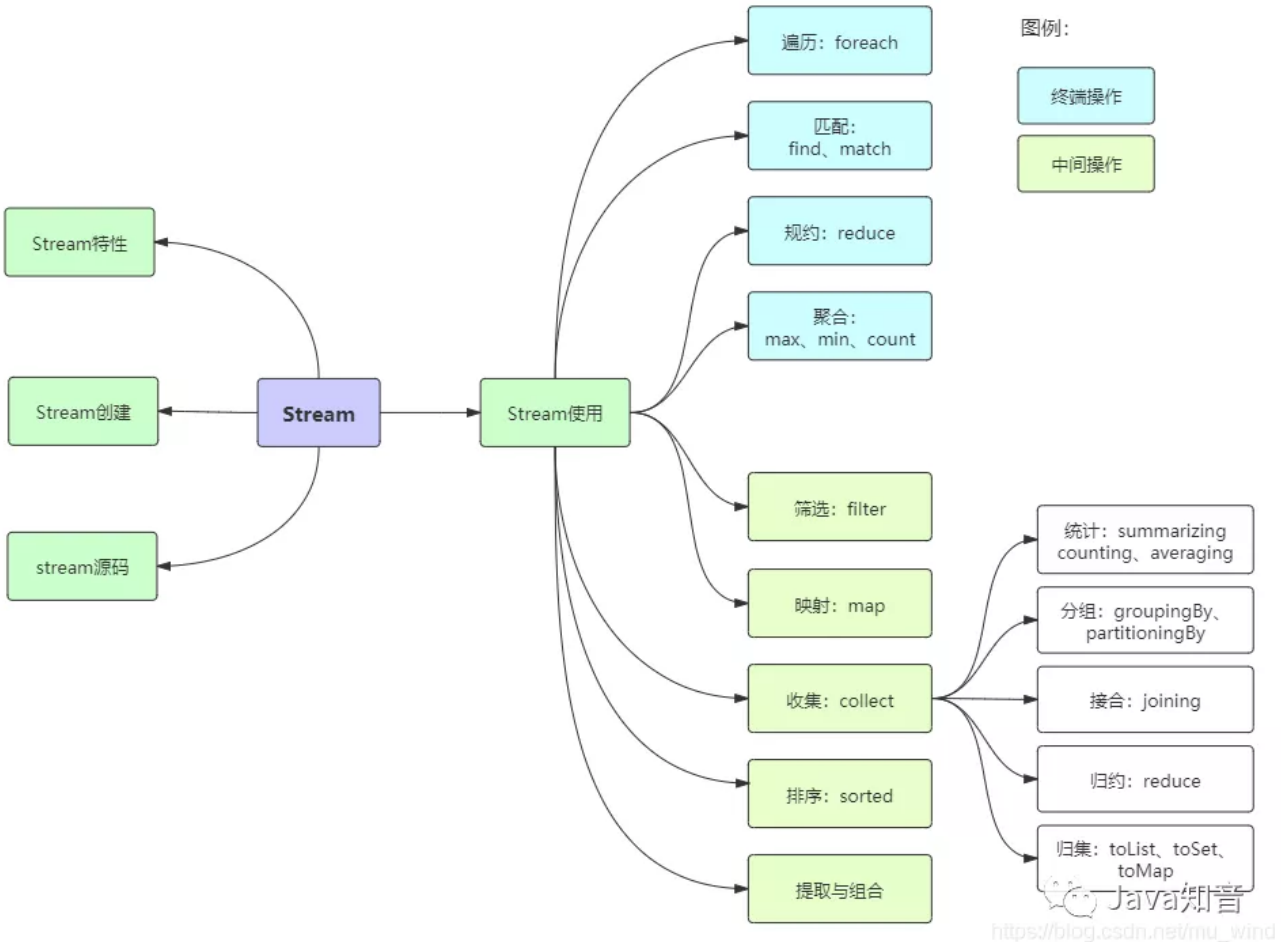
Java8 Stream：2万字20个实例，玩转集合的筛选、归约、分组、聚合

云深i不知处 Java知音 今天



作者：云深i不知处

blog.csdn.net/mu_wind/article/details/109516995



先贴上几个案例，水平高超的同学可以挑战一下：

1. 从员工集合中筛选出salary大于8000的员工，并放置到新的集合里。

2. 统计员工的最高薪资、平均薪资、薪资之和。

3. 将员工按薪资从高到低排序，同样薪资者年龄小者在前。

4. 将员工按性别分类，将员工按性别和地区分类，将员工按薪资是否高于8000分为两部分。

用传统的迭代处理也不是很难，但代码就显得冗余了，跟Stream相比高下立判。

1 Stream概述

Java 8 是一个非常成功的版本，这个版本新增的Stream，配合同版本出现的 Lambda ，给我们操作集合（Collection）提供了极大的便利。

那么什么是Stream?

> Stream将要处理的元素集合看作一种流，在流的过程中，借助Stream API对流中的元素进行操作，比如：筛选、排序、聚合等。

Stream可以由数组或集合创建，对流的操作分为两种：

1. 中间操作，每次返回一个新的流，可以有多个。
2. 终端操作，每个流只能进行一次终端操作，终端操作结束后流无法再次使用。终端操作会产生一个新的集合或值。

另外，Stream有几个特性：

1. stream不存储数据，而是按照特定的规则对数据进行计算，一般会输出结果。
2. stream不会改变数据源，通常情况下会产生一个新的集合或一个值。
3. stream具有延迟执行特性，只有调用终端操作时，中间操作才会执行。

2 Stream的创建

Stream可以通过集合数组创建。

1、通过 java.util.Collection.stream() 方法用集合创建流

```
List<String> list = Arrays.asList("a", "b", "c");
// 创建一个顺序流
```

```
Stream<String> stream = list.stream();
// 创建一个并行流
Stream<String> parallelStream = list.parallelStream();
```

2、使用java.util.Arrays.stream(T[] array)方法用数组创建流

```
int[] array={1,3,5,6,8};
IntStream stream = Arrays.stream(array);
```

3、使用Stream的静态方法：of()、iterate()、generate()

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);

Stream<Integer> stream2 = Stream.iterate(0, (x) -> x + 3).limit(4);
stream2.forEach(System.out::println); // 0 2 4 6 8 10

Stream<Double> stream3 = Stream.generate(Math::random).limit(3);
stream3.forEach(System.out::println);
```

输出结果：

> 0 3 6 9
> 0.6796156909271994
> 0.1914314208854283
> 0.8116932592396652

stream和parallelStream的简单区分： stream是顺序流，由主线程按顺序对流执行操作，而parallelStream是并行流，内部以多线程并行执行的方式对流进行操作，但前提是流中的数据处理没有顺序要求。例如筛选集合中的奇数，两者的处理不同之处：

如果流中的数据量足够大，并行流可以加快处速度。

除了直接创建并行流，还可以通过parallel()把顺序流转换成并行流：

```
Optional<Integer> findFirst = list.stream().parallel().filter(x->x>6).findFirst();
```

3 Stream的使用

在使用stream之前，先理解一个概念：Optional 。

> Optional类是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。
> 更详细说明请见：https://www.runoob.com/java/java8-optional-class.html

接下来，大批代码向你袭来！我将用20个案例将Stream的使用整得明明白白，只要跟着敲一遍代码，就能很好地掌握。

案例使用的员工类

这是后面案例中使用的员工类：

```
List<Person> personList = new ArrayList<Person>();
personList.add(new Person("Tom", 8900, "male", "New York"));
personList.add(new Person("Jack", 7000, "male", "Washington"));
personList.add(new Person("Lily", 7800, "female", "Washington"));
personList.add(new Person("Anni", 8200, "female", "New York"));
personList.add(new Person("Owen", 9500, "male", "New York"));
personList.add(new Person("Alisa", 7900, "female", "New York"));

class Person {
 private String name;  // 姓名
 private int salary; // 薪资
 private int age; // 年龄
 private String sex; //性别
 private String area;  // 地区

 // 构造方法
 public Person(String name, int salary, int age,String sex,String area) {
  this.name = name;
  this.salary = salary;
  this.age = age;
  this.sex = sex;
  this.area = area;
 }
 // 省略了get和set，请自行添加
```

```
}
```

3.1 遍历/匹配（foreach/find/match）

Stream也是支持类似集合的遍历和匹配元素的，只是Stream中的元素是以Optional类型存在的。Stream的遍历、匹配非常简单。

```
// import已省略，请自行添加，后面代码亦是

public class StreamTest {
 public static void main(String[] args) {
        List<Integer> list = Arrays.asList(7, 6, 9, 3, 8, 2, 1);

        // 遍历输出符合条件的元素
        list.stream().filter(x -> x > 6).forEach(System.out::println);
        // 匹配第一个
        Optional<Integer> findFirst = list.stream().filter(x -> x > 6).findFirst();
        // 匹配任意（适用于并行流）
        Optional<Integer> findAny = list.parallelStream().filter(x -> x > 6).findAny();
        // 是否包含符合特定条件的元素
        boolean anyMatch = list.stream().anyMatch(x -> x < 6);
        System.out.println("匹配第一个值：" + findFirst.get());
        System.out.println("匹配任意一个值：" + findAny.get());
        System.out.println("是否存在大于6的值：" + anyMatch);
    }
}
```

3.2 筛选（filter）

筛选，是按照一定的规则校验流中的元素，将符合条件的元素提取到新的流中的操作。

案例一：筛选出Integer集合中大于7的元素，并打印出来

```
public class StreamTest {
 public static void main(String[] args) {
  List<Integer> list = Arrays.asList(6, 7, 3, 8, 1, 2, 9);
  Stream<Integer> stream = list.stream();
  stream.filter(x -> x > 7).forEach(System.out::println);
 }
}
```

预期结果：

> 8 9

案例二：筛选员工中工资高于8000的人，并形成新的集合。 形成新集合依赖collect（收集），后文有详细介绍。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));
  personList.add(new Person("Anni", 8200, 24, "female", "New York"));
  personList.add(new Person("Owen", 9500, 25, "male", "New York"));
  personList.add(new Person("Alisa", 7900, 26, "female", "New York"));
```

```
  List<String> fiterList = personList.stream().filter(x -> x.getSalary() > 8000).map(Person::getN
    .collect(Collectors.toList());
  System.out.print("高于8000的员工姓名：" + fiterList);
 }
}
```

运行结果：

> 高于8000的员工姓名：[Tom, Anni, Owen]

3.3 聚合（max/min/count)

max、min、count这些字眼你一定不陌生，没错，在mysql中我们常用它们进行数据统计。Java stream中也引入了这些概念和用法，极大地方便了我们对集合、数组的数据统计工作。

案例一：获取String集合中最长的元素。

```
public class StreamTest {
 public static void main(String[] args) {
  List<String> list = Arrays.asList("adnm", "admmt", "pot", "xbangd", "weoujgsd");

  Optional<String> max = list.stream().max(Comparator.comparing(String::length));
  System.out.println("最长的字符串：" + max.get());
 }
}
```

输出结果：

> 最长的字符串：weoujgsd

案例二：获取Integer集合中的最大值。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Integer> list = Arrays.asList(7, 6, 9, 4, 11, 6);

  // 自然排序
  Optional<Integer> max = list.stream().max(Integer::compareTo);
  // 自定义排序
  Optional<Integer> max2 = list.stream().max(new Comparator<Integer>() {
   @Override
   public int compare(Integer o1, Integer o2) {
    return o1.compareTo(o2);
   }
  });
  System.out.println("自然排序的最大值：" + max.get());
  System.out.println("自定义排序的最大值：" + max2.get());
 }
}
```

输出结果：

> 自然排序的最大值：11
> 自定义排序的最大值：11

案例三：获取员工工资最高的人。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));
  personList.add(new Person("Anni", 8200, 24, "female", "New York"));
```

```
  personList.add(new Person("Owen", 9500, 25, "male", "New York"));
  personList.add(new Person("Alisa", 7900, 26, "female", "New York"));

  Optional<Person> max = personList.stream().max(Comparator.comparingInt(Person::getSalary));
  System.out.println("员工工资最大值：" + max.get().getSalary());
 }
}
```

输出结果：

> 员工工资最大值：9500

案例四：计算Integer集合中大于6的元素的个数。

```
import java.util.Arrays;
import java.util.List;

public class StreamTest {
 public static void main(String[] args) {
  List<Integer> list = Arrays.asList(7, 6, 4, 8, 2, 11, 9);

  long count = list.stream().filter(x -> x > 6).count();
  System.out.println("list中大于6的元素个数：" + count);
 }
}
```

输出结果：

> list中大于6的元素个数：4

3.4 映射(map/flatMap)

映射，可以将一个流的元素按照一定的映射规则映射到另一个流中。分为map和flatMap：

- map：接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。

- flatMap：接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

案例一：英文字符串数组的元素全部改为大写。整数数组每个元素+3。

```
public class StreamTest {
 public static void main(String[] args) {
  String[] strArr = { "abcd", "bcdd", "defde", "fTr" };
  List<String> strList = Arrays.stream(strArr).map(String::toUpperCase).collect(Collectors.toList

  List<Integer> intList = Arrays.asList(1, 3, 5, 7, 9, 11);
  List<Integer> intListNew = intList.stream().map(x -> x + 3).collect(Collectors.toList());

  System.out.println("每个元素大写：" + strList);
  System.out.println("每个元素+3：" + intListNew);
 }
}
```

输出结果：

> 每个元素大写：[ABCD, BCDD, DEFDE, FTR]
> 每个元素+3：[4, 6, 8, 10, 12, 14]

案例二：将员工的薪资全部增加1000。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));
  personList.add(new Person("Anni", 8200, 24, "female", "New York"));
  personList.add(new Person("Owen", 9500, 25, "male", "New York"));
  personList.add(new Person("Alisa", 7900, 26, "female", "New York"));

  // 不改变原来员工集合的方式
  List<Person> personListNew = personList.stream().map(person -> {
   Person personNew = new Person(person.getName(), 0, 0, null, null);
   personNew.setSalary(person.getSalary() + 10000);
   return personNew;
  }).collect(Collectors.toList());
  System.out.println("一次改动前：" + personList.get(0).getName() + "-->" + personList.get(0).getS
  System.out.println("一次改动后：" + personListNew.get(0).getName() + "-->" + personListNew.get(0

  // 改变原来员工集合的方式
  List<Person> personListNew2 = personList.stream().map(person -> {
   person.setSalary(person.getSalary() + 10000);
   return person;
  }).collect(Collectors.toList());
  System.out.println("二次改动前：" + personList.get(0).getName() + "-->" + personListNew.get(0).g
  System.out.println("二次改动后：" + personListNew2.get(0).getName() + "-->" + personListNew.get(
 }
}
```

输出结果：

> 一次改动前：Tom->8900
> 一次改动后：Tom->18900
> 二次改动前：Tom->18900
> 二次改动后：Tom->18900

案例三：将两个字符数组合并成一个新的字符数组。

```
public class StreamTest {
 public static void main(String[] args) {
  List<String> list = Arrays.asList("m,k,l,a", "1,3,5,7");
  List<String> listNew = list.stream().flatMap(s -> {
   // 将每个元素转换成一个stream
   String[] split = s.split(",");
   Stream<String> s2 = Arrays.stream(split);
   return s2;
  }).collect(Collectors.toList());

  System.out.println("处理前的集合：" + list);
  System.out.println("处理后的集合：" + listNew);
 }
}
```

输出结果：

> 处理前的集合：[m-k-l-a, 1-3-5]
> 处理后的集合：[m, k, l, a, 1, 3, 5]

3.5 归约(reduce)

归约，也称缩减，顾名思义，是把一个流缩减成一个值，能实现对集合求和、求乘积和求最值操作。

案例一：求Integer集合的元素之和、乘积和最大值。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Integer> list = Arrays.asList(1, 3, 2, 8, 11, 4);
  // 求和方式1
  Optional<Integer> sum = list.stream().reduce((x, y) -> x + y);
  // 求和方式2
  Optional<Integer> sum2 = list.stream().reduce(Integer::sum);
  // 求和方式3
  Integer sum3 = list.stream().reduce(0, Integer::sum);

  // 求乘积
  Optional<Integer> product = list.stream().reduce((x, y) -> x * y);

  // 求最大值方式1
  Optional<Integer> max = list.stream().reduce((x, y) -> x > y ? x : y);
  // 求最大值写法2
  Integer max2 = list.stream().reduce(1, Integer::max);

  System.out.println("list求和: " + sum.get() + "," + sum2.get() + "," + sum3);
  System.out.println("list求积: " + product.get());
  System.out.println("list求和: " + max.get() + "," + max2);
 }
}
```

输出结果：

> list求和：29,29,29
> list求积：2112
> list求和：11,11

案例二：求所有员工的工资之和和最高工资。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));
  personList.add(new Person("Anni", 8200, 24, "female", "New York"));
```

```
        personList.add(new Person("Owen", 9500, 25, "male", "New York"));
        personList.add(new Person("Alisa", 7900, 26, "female", "New York"));

        // 求工资之和方式1：
        Optional<Integer> sumSalary = personList.stream().map(Person::getSalary).reduce(Integer::sum);
        // 求工资之和方式2：
        Integer sumSalary2 = personList.stream().reduce(0, (sum, p) -> sum += p.getSalary(),
                (sum1, sum2) -> sum1 + sum2);
        // 求工资之和方式3：
        Integer sumSalary3 = personList.stream().reduce(0, (sum, p) -> sum += p.getSalary(), Integer::su

        // 求最高工资方式1：
        Integer maxSalary = personList.stream().reduce(0, (max, p) -> max > p.getSalary() ? max : p.get
                Integer::max);
        // 求最高工资方式2：
        Integer maxSalary2 = personList.stream().reduce(0, (max, p) -> max > p.getSalary() ? max : p.ge
                (max1, max2) -> max1 > max2 ? max1 : max2);

        System.out.println("工资之和：" + sumSalary.get() + "," + sumSalary2 + "," + sumSalary3);
        System.out.println("最高工资：" + maxSalary + "," + maxSalary2);
    }
}
```

输出结果：

> 工资之和：49300,49300,49300
> 最高工资：9500,9500

3.6 收集(collect)

collect，收集，可以说是内容最繁多、功能最丰富的部分了。从字面上去理解，就是把一个流收集起来，最终可以是收集成一个值也可以收集成一个新的集合。

> collect主要依赖java.util.stream.Collectors类内置的静态方法。

3.6.1 归集(toList/toSet/toMap)

因为流不存储数据，那么在流中的数据完成处理后，需要将流中的数据重新归集到新的集合里。toList、toSet和toMap比较常用，另外还有toCollection、toConcurrentMap等复杂一些的用法。

下面用一个案例演示toList、toSet和toMap：

```
public class StreamTest {
 public static void main(String[] args) {
  List<Integer> list = Arrays.asList(1, 6, 3, 4, 6, 7, 9, 6, 20);
  List<Integer> listNew = list.stream().filter(x -> x % 2 == 0).collect(Collectors.toList());
  Set<Integer> set = list.stream().filter(x -> x % 2 == 0).collect(Collectors.toSet());

  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));
  personList.add(new Person("Anni", 8200, 24, "female", "New York"));

  Map<?, Person> map = personList.stream().filter(p -> p.getSalary() > 8000)
    .collect(Collectors.toMap(Person::getName, p -> p));
  System.out.println("toList:" + listNew);
  System.out.println("toSet:" + set);
  System.out.println("toMap:" + map);
 }
}
```

运行结果：

> toList：[6, 4, 6, 6, 20]
> toSet：[4, 20, 6]
> toMap：{Tom=mutest.Person@5fd0d5ae, Anni=mutest.Person@2d98a335}

3.6.2 统计(count/averaging)

Collectors提供了一系列用于数据统计的静态方法：

- 计数：count

- 平均值：averagingInt、averagingLong、averagingDouble
- 最值：maxBy、minBy
- 求和：summingInt、summingLong、summingDouble
- 统计以上所有：summarizingInt、summarizingLong、summarizingDouble

案例：统计员工人数、平均工资、工资总额、最高工资。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));

  // 求总数
  Long count = personList.stream().collect(Collectors.counting());
  // 求平均工资
  Double average = personList.stream().collect(Collectors.averagingDouble(Person::getSalary));
  // 求最高工资
  Optional<Integer> max = personList.stream().map(Person::getSalary).collect(Collectors.maxBy(Inte
  // 求工资之和
  Integer sum = personList.stream().collect(Collectors.summingInt(Person::getSalary));
  // 一次性统计所有信息
  DoubleSummaryStatistics collect = personList.stream().collect(Collectors.summarizingDouble(Perso

  System.out.println("员工总数：" + count);
  System.out.println("员工平均工资：" + average);
  System.out.println("员工工资总和：" + sum);
  System.out.println("员工工资所有统计：" + collect);
 }
}
```

运行结果：

```
员工总数：3
员工平均工资：7900.0
```

> 员工工资总和：23700
> 员工工资所有统计：DoubleSummaryStatistics{count=3, sum=23700.000000,min=7000.000000, average=7900.000000, max=8900.000000}

3.6.3 分组(partitioningBy/groupingBy)

- 分区：将stream按条件分为两个Map，比如员工按薪资是否高于8000分为两部分。
- 分组：将集合分为多个Map，比如员工按性别分组。有单级分组和多级分组。

案例：将员工按薪资是否高于8000分为两部分；将员工按性别和地区分组

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, "male", "New York"));
  personList.add(new Person("Jack", 7000, "male", "Washington"));
  personList.add(new Person("Lily", 7800, "female", "Washington"));
  personList.add(new Person("Anni", 8200, "female", "New York"));
  personList.add(new Person("Owen", 9500, "male", "New York"));
  personList.add(new Person("Alisa", 7900, "female", "New York"));

  // 将员工按薪资是否高于8000分组
        Map<Boolean, List<Person>> part = personList.stream().collect(Collectors.partitioningBy(x
        // 将员工按性别分组
        Map<String, List<Person>> group = personList.stream().collect(Collectors.groupingBy(Perso
        // 将员工先按性别分组，再按地区分组
        Map<String, Map<String, List<Person>>> group2 = personList.stream().collect(Collectors.gr
        System.out.println("员工按薪资是否大于8000分组情况：" + part);
        System.out.println("员工按性别分组情况：" + group);
        System.out.println("员工按性别、地区：" + group2);
 }
}
```

输出结果：

```
员工按薪资是否大于8000分组情况：{false=[mutest.Person@2d98a335, mutest.Person@16b98e56, mutest.Perso
员工按性别分组情况：{female=[mutest.Person@16b98e56, mutest.Person@4f3f5b24, mutest.Person@7ef20235
员工按性别、地区：{female={New York=[mutest.Person@4f3f5b24, mutest.Person@7ef20235], Washington=[m
```

3.6.4 接合(joining)

joining可以将stream中的元素用特定的连接符（没有的话，则直接连接）连接成一个字符串。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));

  String names = personList.stream().map(p -> p.getName()).collect(Collectors.joining(","));
  System.out.println("所有员工的姓名：" + names);
  List<String> list = Arrays.asList("A", "B", "C");
  String string = list.stream().collect(Collectors.joining("-"));
  System.out.println("拼接后的字符串：" + string);
 }
}
```

运行结果：

> 所有员工的姓名：Tom,Jack,Lily
> 拼接后的字符串：A-B-C

3.6.5 归约(reducing)

Collectors类提供的reducing方法，相比于stream本身的reduce方法，增加了对自定义归约的支持。

```
public class StreamTest {
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();
  personList.add(new Person("Tom", 8900, 23, "male", "New York"));
  personList.add(new Person("Jack", 7000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 7800, 21, "female", "Washington"));

  // 每个员工减去起征点后的薪资之和（这个例子并不严谨，但一时没想到好的例子）
  Integer sum = personList.stream().collect(Collectors.reducing(0, Person::getSalary, (i, j) -> (
  System.out.println("员工扣税薪资总和：" + sum);

  // stream的reduce
  Optional<Integer> sum2 = personList.stream().map(Person::getSalary).reduce(Integer::sum);
  System.out.println("员工薪资总和：" + sum2.get());
 }
}
```

运行结果：

> 员工扣税薪资总和：8700
> 员工薪资总和：23700

3.7 排序(sorted)

sorted，中间操作。有两种排序：

- sorted()：自然排序，流中元素需实现Comparable接口
- sorted(Comparator com)：Comparator排序器自定义排序

案例：将员工按工资由高到低（工资一样则按年龄由大到小）排序

```
public class StreamTest {
```

```
 public static void main(String[] args) {
  List<Person> personList = new ArrayList<Person>();

  personList.add(new Person("Sherry", 9000, 24, "female", "New York"));
  personList.add(new Person("Tom", 8900, 22, "male", "Washington"));
  personList.add(new Person("Jack", 9000, 25, "male", "Washington"));
  personList.add(new Person("Lily", 8800, 26, "male", "New York"));
  personList.add(new Person("Alisa", 9000, 26, "female", "New York"));

  // 按工资增序排序
  List<String> newList = personList.stream().sorted(Comparator.comparing(Person::getSalary)).map(P
    .collect(Collectors.toList());
  // 按工资倒序排序
  List<String> newList2 = personList.stream().sorted(Comparator.comparing(Person::getSalary).reve
    .map(Person::getName).collect(Collectors.toList());
  // 先按工资再按年龄自然排序（从小到大）
  List<String> newList3 = personList.stream().sorted(Comparator.comparing(Person::getSalary).reve
    .map(Person::getName).collect(Collectors.toList());
  // 先按工资再按年龄自定义排序（从大到小）
  List<String> newList4 = personList.stream().sorted((p1, p2) -> {
   if (p1.getSalary() == p2.getSalary()) {
    return p2.getAge() - p1.getAge();
   } else {
    return p2.getSalary() - p1.getSalary();
   }
  }).map(Person::getName).collect(Collectors.toList());

  System.out.println("按工资自然排序：" + newList);
  System.out.println("按工资降序排序：" + newList2);
  System.out.println("先按工资再按年龄自然排序：" + newList3);
  System.out.println("先按工资再按年龄自定义降序排序：" + newList4);
 }
}
```

运行结果：

> 按工资自然排序：[Lily, Tom, Sherry, Jack, Alisa]
> 按工资降序排序：[Sherry, Jack, Alisa,Tom, Lily]
> 先按工资再按年龄自然排序：[Sherry, Jack, Alisa, Tom, Lily]
> 先按工资再按年龄自定义降序排序：[Alisa, Jack, Sherry, Tom, Lily]

3.8 提取/组合

流也可以进行合并、去重、限制、跳过等操作。


```
public class StreamTest {
 public static void main(String[] args) {
  String[] arr1 = { "a", "b", "c", "d" };
  String[] arr2 = { "d", "e", "f", "g" };

  Stream<String> stream1 = Stream.of(arr1);
  Stream<String> stream2 = Stream.of(arr2);
  // concat:合并两个流 distinct：去重
  List<String> newList = Stream.concat(stream1, stream2).distinct().collect(Collectors.toList());
```

```
    // limit：限制从流中获得前n个数据
    List<Integer> collect = Stream.iterate(1, x -> x + 2).limit(10).collect(Collectors.toList());
    // skip：跳过前n个数据
    List<Integer> collect2 = Stream.iterate(1, x -> x + 2).skip(1).limit(5).collect(Collectors.toLi

    System.out.println("流合并：" + newList);
    System.out.println("limit：" + collect);
    System.out.println("skip：" + collect2);
  }
}
```

运行结果：

> 流合并：[a, b, c, d, e, f, g]
> limit：[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
> skip：[3, 5, 7, 9, 11]

4 Stream源码解读

这部分等有时间慢慢分解吧。

好，以上就是全部内容，能坚持看到这里，你一定很有收获，那么动一动拿offer的小手，点个赞再走吧。

END

推荐好文

强大，10k+点赞的 SpringBoot 后台管理系统竟然出了详细教程！

为什么MySQL不推荐使用uuid或者雪花id作为主键？

为什么建议大家使用 Linux 开发？爽（外加七个感叹号）

IntelliJ IDEA 15款 神级超级牛逼插件推荐（自用，真的超级牛逼）

炫酷，SpringBoot+Echarts实现用户访问地图可视化（附源码）

记一次由Redis分布式锁造成的重大事故，避免以后踩坑！

十分钟学会使用 Elasticsearch 优雅搭建自己的搜索系统（附源码）

喜欢此内容的人还喜欢

SQL 编程思想：一切皆关系

Java面试题精选

分库分表之 Sharding-JDBC 中间件，看这篇真的够了!

码海

List去除重复数据的五种方式

java版web项目