# **创建流**

parallelStream()方法能够充分利用多核CPU的优势，使用多线程加快对集合数据的处理速度

## **1、Stream创建**

Stream<Integer> stream1 = Stream.of(1,2,3,4,5);

1. **Collection集合创建**

parallelStream()方法也可以创建流，并能够充分利用多核CPU的优势，使用多线程加快对集合数据的处理速度

List<Integer> integerList = new ArrayList<>();

integerList.add(1);

integerList.add(2);

integerList.add(3);

integerList.add(4);

integerList.add(5);

Stream<Integer> listStream = integerList.stream();

**3、Array数组创建**

int[] intArr = {1, 2, 3, 4, 5};

IntStream arrayStream = 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**

**注：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**

## 4、文件创建

try {

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

} catch (IOException e) {

e.printStackTrace();

}

**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、函数创建

iterator

Stream<Integer> iterateStream = 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> generateStream = 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**

# **操作流**

# **中间操作符**

**中间操作不会立即执行，只有等到终端操作的时候，流才开始真正地遍历，用于映射、过滤等。通俗点说，就是一次遍历执行多个操作，性能就大大提高了。**

**一个流可以跟随零个或多个中间操作。**

|  |  |  |
| --- | --- | --- |
| **流方法** | **含义** | **示例** |
| filter | 用于通过设置的条件过滤出元素 | List strings = Arrays.asList(“abc”, “”, “bc”, “efg”, “abcd”,"", “jkl”);  List filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList()); |
| map | 接受一个函数作为参数。这个函数会被应用到每个元素上，并将其映射成一个新的元素（使用映射一词，是因为它和转换类似，但其中的细微差别在于它是“创建一个新版本”而不是去“修改”） | List strings = Arrays.asList(“abc”, “abc”, “bc”, “efg”, “abcd”,“jkl”, “jkl”);  List mapped = strings.stream().map(str->str+"-itcast").collect(Collectors.toList()); |
| distinct | 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 | List numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);numbers.stream().filter(i -> i % 2 == 0).distinct().forEach(System.out::println); |
| sorted | 返回排序后的流 | List strings1 = Arrays.asList(“abc”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  List sorted1 = strings1.stream().sorted().collect(Collectors.toList()); |
| limit | 会返回一个不超过给定长度的流 | List strings = Arrays.asList(“abc”, “abc”, “bc”, “efg”, “abcd”,“jkl”, “jkl”);  List limited = strings.stream().limit(3).collect(Collectors.toList()); |
| skip | 返回一个扔掉了前n个元素的流 | List strings = Arrays.asList(“abc”, “abc”, “bc”, “efg”, “abcd”,“jkl”, “jkl”);  List skiped = strings.stream().skip(3).collect(Collectors.toList()); |
| flatMap | 使用flatMap方法的效果是，各个数组并不是分别映射成一个流，而是映射成流的内容。所有使用map(Arrays::stream)时生成的单个流都被合并起来，即扁平化为一个流 | List strings = Arrays.asList(“abc”, “abc”, “bc”, “efg”, “abcd”,“jkl”, “jkl”);  Stream flatMap = strings.stream().flatMap(Java8StreamTest::getCharacterByString); |
| peek | 对元素进行遍历处理 | List strings = Arrays.asList(“abc”, “abc”, “bc”, “efg”, “abcd”,“jkl”, “jkl”);  strings .stream().peek(str-> str + "a").forEach(System.out::println); |

# **终端操作符**

**Stream流执行完终端操作之后，无法再执行其他动作，否则会报状态异常，提示该流已经被执行操作或者被关闭，想要再次执行操作必须重新创建Stream流。**

**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**

|  |  |  |
| --- | --- | --- |
| **流方法** | **含义** | **示例** |
| collect | 收集器，将流转换为其他形式 | List strings = Arrays.asList(“cv”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  Set set = strings.stream().collect(Collectors.toSet());  List list = strings.stream().collect(Collectors.toList());  Map<String, String> map = strings.stream().collect(Collectors.toMap(v ->v.concat("\_name"), v1 -> v1, (v1, v2) -> v1)); |
| forEach | 遍历流 | List strings = Arrays.asList(“cv”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);strings.stream().forEach(s -> out.println(s)); |
| findFirst | 返回第一个元素 | List strings = Arrays.asList(“cv”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  Optional first = strings.stream().findFirst(); |
| findAny | 将返回当前流中的任意元素 | List strings = Arrays.asList(“cv”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  Optional any = strings.stream().findAny(); |
| count | 返回流中元素总数 | List strings = Arrays.asList(“cv”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  long count = strings.stream().count(); |
| sum | 求和 | int sum = userList.stream().mapToInt(User::getId).sum(); |
| max | 最大值 | int max = userList.stream().max(Comparator.comparingInt(User::getId)).get().getId(); |
| min | 最小值 | int min = userList.stream().min(Comparator.comparingInt(User::getId)).get().getId(); |
| anyMatch | 只要有一个元素匹配传入的条件，就返回 true | List strings = Arrays.asList(“abc”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  boolean b = strings.stream().anyMatch(s -> s == “abc”); |
| allMatch | 只有有一个元素不匹配传入的条件，就返回 false；如果全部匹配，则返回true。 | List strings = Arrays.asList(“abc”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  boolean b = strings.stream().allMatch(s -> s == “abc”); |
| noneMatch | 只要有一个元素匹配传入的条件，就返回 false；如果全不匹配，则返回 true。 | List strings = Arrays.asList(“abc”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  boolean b = strings.stream().noneMatch(s -> s == “abc”); |
| reduce | 可以将流中元素反复结合起来，得到一个值 | List strings = Arrays.asList(“cv”, “abd”, “aba”, “efg”, “abcd”,“jkl”, “jkl”);  Optional reduce = strings.stream().reduce((acc,item) -> {return acc+item;});if(reduce.isPresent())out.println(reduce.get()); |

# **Collect收集**

****Collector****：结果收集策略的核心接口，具备将指定元素累加存放到结果容器中的能力；并在Collectors工具中提供了Collector接口的实现类

### **1、toList**

将用户ID存放到List集合中

List<Integer> idList = userList.stream().map(User::getId).collect(Collectors.toList()) ;

### **2、toMap**

将用户ID和Name以Key-Value形式存放到Map集合中

Map<Integer,String> userMap = userList.stream().collect(Collectors.toMap(User::getId,User::getName));

### **3、toSet**

将用户所在城市存放到Set集合中

Set<String> citySet = userList.stream().map(User::getCity).collect(Collectors.toSet());

### **4、counting**

符合条件的用户总数

long count = userList.stream().filter(user -> user.getId()>1).collect(Collectors.counting());

### **5、summingInt**

对结果元素即用户ID求和

Integer sumInt = userList.stream().filter(user -> user.getId()>2).collect(Collectors.summingInt(User::getId)) ;

### ****6、minBy****

筛选元素中ID最小的用户

User maxId = userList.stream().collect(Collectors.minBy(Comparator.comparingInt(User::getId))).get() ;

### **7、joining**

将用户所在城市，以指定分隔符链接成字符串；

String joinCity = userList.stream().map(User::getCity).collect(Collectors.joining("||"));

### **8、groupingBy**

按条件分组，以城市对用户进行分组；

Map<String,List<User>> groupCity =userList.stream().collect(Collectors.groupingBy(User::getCity));

# **排序**

## 基础写法

userList.stream()

.sorted(Comparator.comparing(User::getAge))

.forEach(System.out::println);

## 倒序

写法一

userList= userList.stream()

.sorted(Comparator.comparing(User::getAge).reversed())

.forEach(System.out::println);

写法二

userList= userList.stream()

.sorted(Comparator.comparing(User::getAge, Comparator.reverseOrder()))

.forEach(System.out::println);

Comparator.reverseOrder()参数是让某个条件进行倒序排序.  
reversed()方法是让目前的排序结果进行倒序。

## 多字段排序

userList= userList.stream()

.sorted(Comparator.comparing(User::getAge)).thenComparing(User::getSex))

.forEach(System.out::println);

# **分组**

## **按属性给对象分组**

例子：按性别给用户分组

Map<String, List<User>> sexAndUserMap = UserList.parallelStream()

.collect(Collectors.groupingBy(User::getSex));

## **按A属性分组统计B属性List（Set）**

按性别分组统计名称List

Map<String, List<String>> sexAndNameListMap = UserList.parallelStream()

.collect(Collectors.groupingBy(User::getSex, Collectors.mapping(User::getName, Collectors.toList())));

按性别分组统计名称Set

Map<String, Set<String>> sexAndNameSetMap = UserList.parallelStream()

.collect(Collectors.groupingBy(User::getSex, Collectors.mapping(User::getName, Collectors.toSet())));

## **按A属性分组统计B属性和对象本身**

按性别分组统计名称以及用户本身。

Map<String, Map<String, User>> sexAndNameAndDataMap = UserList.parallelStream()

.collect(Collectors.groupingBy(User::getSex, Collectors.toMap(User::getName, data -> data)));

这里需要确保B属性在A属性下是唯一的，

示例中得保证名称在各性别组下是唯一的否则会产生用户覆盖。如果不是唯一的，可以采用多级分组

## **多级分组，按A属性分组，在此基础上按B属性分组**

Map<String, Map<String, List<User>>> SexAndNameAndUserListMap = UserList.parallelStream()

.collect(Collectors.groupingBy(User::getSex, Collectors.groupingBy(User::getName)));

一个实例

Apple apple01 = new Apple(1, "red", 100, "成都");

Apple apple02 = new Apple(2, "red", 99, "成都");

Apple apple03 = new Apple(3, "orange", 120, "西安");

Apple apple04 = new Apple(4, "orange", 80, "天津");

Apple apple05 = new Apple(5, "orange", 80, "深圳");

Apple apple06 = new Apple(6, "black", 102, "成都");

Apple apple07 = new Apple(7, "black", 99, "成都");

List<Apple> apples = new ArrayList<>();

Collections.addAll(apples, apple01, apple02, apple03, apple04, apple05, apple06, apple07);

//先按照奇数偶数分组，再按照地名分组

Map<String, Map<String, List<Apple>>> collect = apples.stream()

.collect(Collectors.groupingBy(x -> {

if ((x.getId() & 1) == 1) {

return "奇数";

} else {

return "偶数";

}

}, Collectors.groupingBy(Apple::getOroigin)));

System.out.println(collect);

打印如下

{

偶数 = {

成都 = [

Apple {id = 2, colour = 'red', weight = 99, oroigin = '成都'},

Apple {id = 6, colour = 'black', weight = 102, oroigin = '成都'}

],

天津 = [

Apple {id = 4, colour = 'orange', weight = 80, oroigin = '天津'}

]

},

奇数 = {

成都 = [

Apple {id = 1, colour = 'red', weight = 100, oroigin = '成都'},

Apple {id = 7, colour = 'black', weight = 99, oroigin = '成都'}

],

西安 = [

Apple {id = 3, colour = 'orange', weight = 120, oroigin = '西安'}

],

深圳 = [

Apple {id = 5, colour = 'orange', weight = 80, oroigin = '深圳'}

]

}

}

## **遍历对象拿A和B属性建立Map映射关系**

遍历用户，构建一个名称与性别的Map映射关系

Map<String, String> nameAndSexMap = UserList.parallelStream()

.collect(Collectors.toMap(User::getName, User::getSex));

这里需要确保A在全局下是唯一的，否则会出现B属性值覆盖的情况