

目录	
第1章 基础	
01 开篇词：为什么学习本专栏	
02 String、Long 源码解析和面试题	
03 Java 常用关键字理解	
04 Arrays、Collections、Objects 常用方法源码解析	
第2章 集合	
05 ArrayList 源码解析和设计思路	
06 LinkedList 源码解析	
07 List 源码会问哪些面试题	
08 HashMap 源码解析	
09 TreeMap 和 LinkedHashMap 核心源码解析	
10 Map源码会问哪些面试题	
11 HashSet、TreeSet 源码解析	
12 彰显细节：看集合源码对我们实际工作的帮助和应用	
13 差异对比：集合在 Java 7 和 8 有何不同和改进	
14 简化工作：Guava Lists Maps 实际工作运用和源码	
第3章 并发集合类	
15 CopyOnWriteArrayList 源码解析和设计思路	
16 ConcurrentHashMap 源码解析和设计思路	
17 并发 List、Map源码面试题	
18 场景集合：并发 List、Map的应用	

42 常用的 Lambda 表达式使用场景解析和应用

更新时间：2019-11-27 10:56:25



“耐心和恒心总会得到报酬的。”
——爱因斯坦

引导语

我们日常工作中，Lambda 使用比较多的场景，就是 List 或 Map 下的 Lambda 流操作，往往几行代码可以帮助我们实现多层 for 循环嵌套的复杂代码，接下来我们把 Lambda 流的常用方法用案例讲解一下。

1 数据准备

本文演示的所有代码都在 demo.eight.LambdaExpressionDemo 中，首先我们需要准备一些测试的数据，如下：

```
@Data
// 学生数据结构
class StudentDTO implements Serializable {

    private static final long serialVersionUID = -7716352032236707189L;

    public StudentDTO() {
    }

    public StudentDTO(Long id, String code, String name, String sex, Double scope,
        List<Course> learningCourses) {
        this.id = id;
        this.code = code;
        this.name = name;
        this.sex = sex;
        this.scope = scope;
        this.learningCourses = learningCourses;
    }
}
```

目录	<pre>* id */ private Long id; /** * 学号 唯一标识 */ private String code; /** * 学生名字 */ private String name; /** * 性别 */ private String sex; /** * 分数 */ private Double score; /** * 要学习的课程 */ private List<Course> learningCourses; } @Data // 课程数据结构 class Course implements Serializable { private static final long serialVersionUID = 2896201730223729591L; /** * 课程 ID */ private Long id; /** * 课程 name */ private String name; public Course(Long id, String name) { this.id = id; this.name = name; } } // 初始化数据 private final List<StudentDTO> students = new ArrayList<StudentDTO>(){ { // 添加学生数据 add(new StudentDTO(1L,"W199","小美","WM",100D,new ArrayList<Course>(){ { // 添加学生学习的课程 add(new Course(300L,"语文")); add(new Course(301L,"数学")); add(new Course(302L,"英语")); } })); add(new StudentDTO(2L,"W25","小美","WM",100D,Lists.newArrayList());); } }</pre>
----	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

目录

```
        add(new Course(304L,"体育"));
    }
    });
    add(new StudentDTO(4L,"W1","小蓝","M",10D,new ArrayList<Course>(){
        {
            add(new Course(301L,"数学"));
            add(new Course(305L,"美术"));
        }
    });
    }
};
```

请大家稍微看下数据结构，不然看下面案例跑出来的结果会有些吃力。

2 常用方法

2.1 Filter

Filter 为过滤的意思，只要满足 Filter 表达式的数据就可以留下来，不满足的数据被过滤掉，源码如下图：

```
38  /**
39  *
40  * @FunctionalInterface
41  * public interface Predicate<T> {
42  *
43  *     /**
44  *      * Evaluates this predicate on the given element.
45  *      * @param t the input argument
46  *      * @return true if the input argument matches the predicate, otherwise false.
47  *     boolean test(T t);
48  * }
49  */
```

我们写了一个 demo，如下：

```
public void testFilter() {
    // list 在下图中进行了初始化
    List<String> newList = list.stream()
        // 过滤掉我们希望留下来的值
        // StringUtils.equals(str,"hello") 表示我们希望字符串是 hello 能留下来
        // 其他的过滤掉
        .filter(str -> StringUtils.equals(str, "hello"))
        // Collectors.toList() 帮助我们构造最后的返回结果
        .collect(Collectors.toList());
    log.info("TestFilter result is {}", JSON.toJSONString(newList));
}
```

运行结果如下：

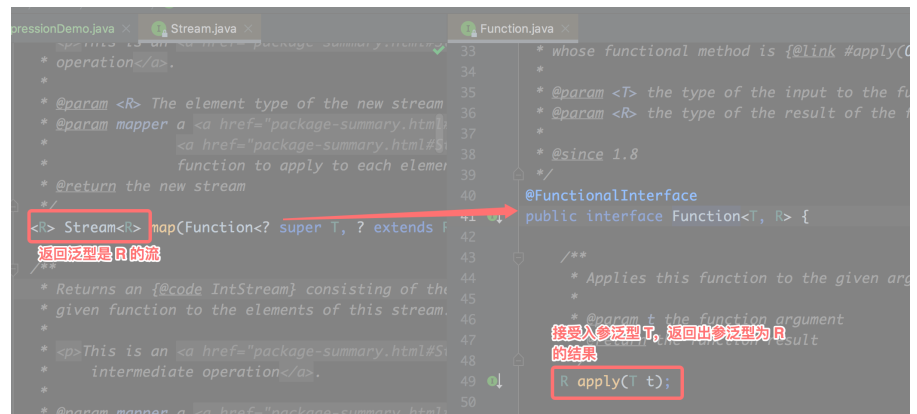
```
private static final List<String> list = ImmutableList.of("hello", "world");
private static final Map<String, String> map = ImmutableMap.of("k1", "v1", "k2", "v2", "k3", "v3");
```

从运行结果看，过滤掉了 world

TestFilter result is ["hello"]

目录

map 方法可以让我们进行一些流的转化，比如原来流中的元素是 A，通过 map 操作，可以使返回的流中的元素是 B，源码如下图：

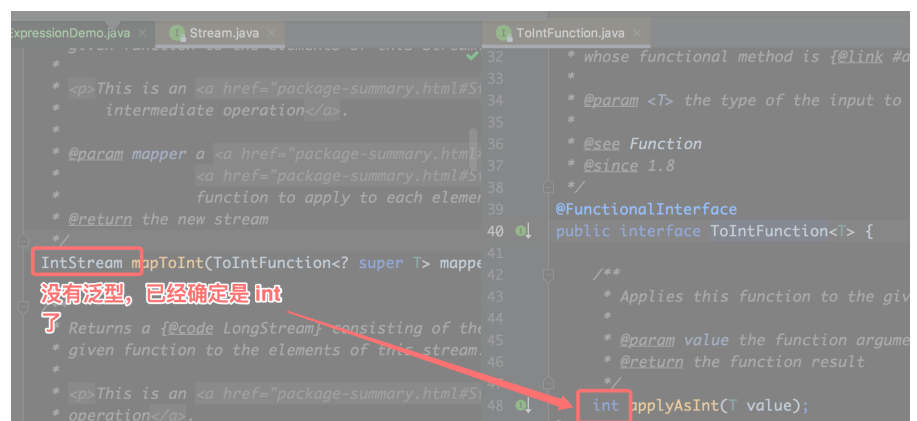


我们写了一个 demo，如下：

```
public void testMap() {
    // 得到所有学生的学号
    // 这里 students.stream() 中的元素是 StudentDTO，通过 map 方法转化成 String 的流
    List<String> codes = students.stream()
        //StudentDTO::getCode 是 s->s.getCode() 的简写
        .map(StudentDTO::getCode)
        .collect(Collectors.toList());
    log.info("TestMap 所有学生的学号为 {}", JSON.toJSONString(codes));
}
// 运行结果为：TestMap 所有学生的学号为 ["W199","W25","W3","W1"]
```

2.3 mapToInt

mapToInt 方法的功能和 map 方法一样，只不过 mapToInt 返回的结果已经没有泛型，已经明确是 int 类型的流了，源码如下：



我们写了一个 demo，如下：

```
public void testMapToInt() {
    List<Integer> ids = students.stream()
        .mapToInt(s -> Integer.valueOf(s.getId() + ""))
        // 一定要有 mapToObj，因为 mapToInt 返回的是 IntStream，因为已经确定是 int 类型了
        // 所有没有泛型的，而 Collectors.toList() 强制要求有泛型的流，所以需要使用 mapToObj
        // 方法返回有泛型的流
        .mapToObj(s -> s)
}
```

目录

```
// 计算学生总分
Double sumScope = students.stream()
    .mapToDouble(s->s.getScope())
    // DoubleStream/IntStream 有许多 sum (求和)、min (求最小值)、max (求最大值)、av
    .sum();
log.info("TestMapToInt 学生总分为: is {}", sumScope);
}
```

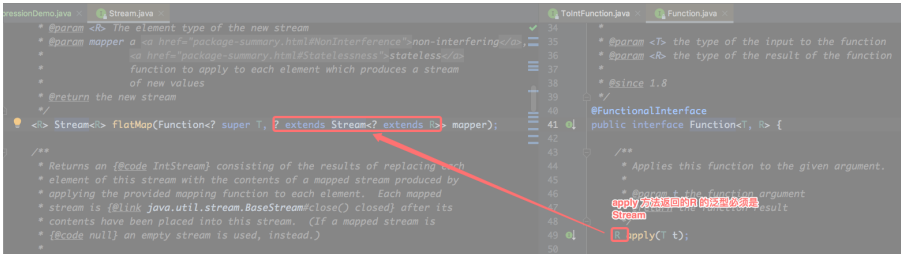
运行结果如下：

TestMapToInt result is [1,2,3,4]

TestMapToInt 学生总分为: is 300.0

2.4 flatMap

flatMap 方法也是可以做一些流的转化，和 map 方法不同的是，其明确了 Function 函数的返回值的泛型是流，源码如下：



写了一个 demo，如下：

```
public void testFlatMap(){
    // 计算学生所有的学习课程，flatMap 返回 List<课程> 格式
    List<Course> courses = students.stream().flatMap(s->s.getLearningCourses().stream())
        .collect(Collectors.toList());
    log.info("TestMapToInt flatMap 计算学生的所有学习课程如下 {}", JSON.toJSONString(courses));

    // 计算学生所有的学习课程，map 返回两层课程嵌套格式
    List<List<Course>> courses2 = students.stream().map(s->s.getLearningCourses())
        .collect(Collectors.toList());
    log.info("TestMapToInt map 计算学生的所有学习课程如下 {}", JSON.toJSONString(courses2));

    List<Stream<Course>> courses3 = students.stream().map(s->s.getLearningCourses().stream())
        .collect(Collectors.toList());
    log.info("TestMapToInt map 计算学生的所有学习课程如下 {}", JSON.toJSONString(courses3));
}
```

运行结果如下：

```
@Test
public void testFlatMap(){
    // 计算学生所有的学习课程，flatMap 返回 List<课程> 格式
    List<Course> courses = students.stream().flatMap(s->s.getLearningCourses().stream())
        .collect(Collectors.toList());
    log.info("TestMapToInt flatMap 计算学生的所有学习课程如下 {}", JSON.toJSONString(courses));

    // 计算学生所有的学习课程，map 返回两层课程嵌套格式
    List<List<Course>> courses2 = students.stream().map(s->s.getLearningCourses())
        .collect(Collectors.toList());
    log.info("TestMapToInt map 计算学生的所有学习课程如下 {}", JSON.toJSONString(courses2));

    List<Stream<Course>> courses3 = students.stream().map(s->s.getLearningCourses().stream())
        .collect(Collectors.toList());
    log.info("TestMapToInt map 计算学生的所有学习课程如下 {}", JSON.toJSONString(courses3));
}
```

```
it flatMap 计算学生的所有学习课程如下 [[{"id":300,"name":"语文"}, {"id":301,"name":"数学"}, {"id":302,"name":"英语"}, {"id":303,"name":"语文"}, {"id":304,"name":"体育"}],
it map 计算学生的所有学习课程如下 [[{"id":300,"name":"语文"}, {"id":301,"name":"数学"}, {"id":302,"name":"英语"}], [{"id":300,"name":"语文"}, {"id":304,"name":"体育"}],
it map 计算学生的所有学习课程如下 [{"parallel":false}, {"parallel":false}, {"parallel":false}, {"parallel":false}]
```

运行结果

}

运行结果如下:

```
- TestSorted 按照学号排序之前 ["W199", "W25", "W3", "W1"]
- TestSorted 按照学号排序之后 is ["W1", "W199", "W25", "W3"]
- TestSorted 按照自然排序 is ["W1", "W199", "W25", "W3"]
- TestSorted 反自然排序 is ["W3", "W25", "W199", "W1"]
```

2.7 peek

peek 方法很简单, 我们在 peek 方法里面做任意没有返回值的事情, 比如打印日志, 如下:

```
students.stream().map(StudentDTO::getCode)
    .peek(s -> log.info("当前循环的学号是{}", s))
    .collect(Collectors.toList());
```

2.8 limit

limit 方法会限制输出值个数, 入参是限制的个数大小, demo 如下:

```
public void testLimit(){
    List<String> beforeCodes = students.stream().map(StudentDTO::getCode).collect(Collectors.toList());
    log.info("TestLimit 限制之前学生的学号为 {}", JSON.toJSONString(beforeCodes));

    List<String> limitCodes = beforeCodes.stream()
        .limit(2L)
        .collect(Collectors.toList());
    log.info("TestLimit 限制最大限制 2 个学生的学号 {}", JSON.toJSONString(limitCodes));

    // 直接连起来写
    List<String> codes = students.stream()
        .map(StudentDTO::getCode)
        .limit(2L)
        .collect(Collectors.toList());
    log.info("TestLimit 限制最大限制 2 个学生的学号 {}", JSON.toJSONString(codes));
}
```

输出结果如下:

```
- TestLimit 限制之前学生的学号为 ["W199", "W25", "W3", "W1"]
- TestLimit 限制最大限制 2 个学生的学号 ["W199", "W25"] 被限制了
- TestLimit 限制最大限制 2 个学生的学号 ["W199", "W25"]
```

2.9 reduce

reduce 方法允许我们在循环里面叠加计算值, 我们写了 demo 如下:

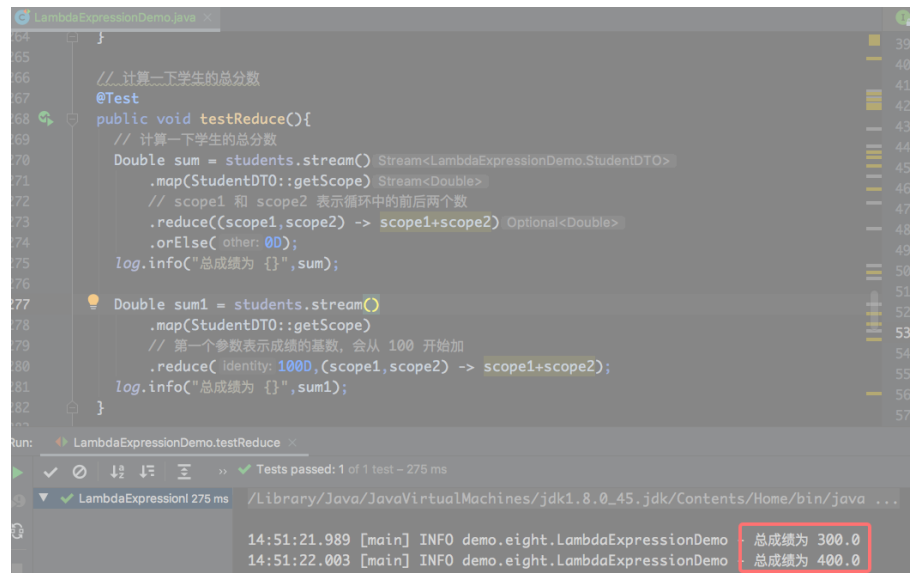
```
public void testReduce(){
    // 计算一下学生的总分数
    Double sum = students.stream()
        .map(StudentDTO::getScore)
        .reduce(0, (sum, score) -> sum + score);
}
```

目录

```
log.info("总成绩为 {}",sum);

Double sum1 = students.stream()
    .map(StudentDTO::getScope)
    // 第一个参数表示成绩的基数, 会从 100 开始加
    .reduce(100D,(scope1,scope2) -> scope1+scope2);
log.info("总成绩为 {}",sum1);
}
```

运行结果如下:



```
65
66 // 计算一下学生的总分数
67 @Test
68 public void testReduce(){
69     // 计算一下学生的总分数
70     Double sum = students.stream().map(StudentDTO::getScope).reduce(0D,(scope1,scope2) -> scope1+scope2);
71     log.info("总成绩为 {}",sum);
72
73     Double sum1 = students.stream().map(StudentDTO::getScope).reduce(100D,(scope1,scope2) -> scope1+scope2);
74     log.info("总成绩为 {}",sum1);
75 }
76
77 Run: LambdaExpressionDemo.testReduce
78 Tests passed: 1 of 1 test - 275 ms
79 /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java ...
80 14:51:21.989 [main] INFO demo.eight.LambdaExpressionDemo 总成绩为 300.0
81 14:51:22.003 [main] INFO demo.eight.LambdaExpressionDemo 总成绩为 400.0
```

第二个计算出来的总成绩多了 100，是因为第二个例子中 reduce 是从基数 100 开始累加的。

2.10 findFirst

findFirst 表示匹配到第一个满足条件的值就返回，demo 如下：

```
// 找到第一个叫小美同学的 ID
@Test
public void testFindFirst(){
    Long id = students.stream()
        .filter(s->StringUtils.equals(s.getName(),"小美"))
        // 同学中有两个叫小美的，这里匹配到第一个就返回
        .findFirst()
        .get().getId();

    log.info("testFindFirst 小美同学的 ID {}",id);

    // 防止空指针
    Long id2 = students.stream()
        .filter(s->StringUtils.equals(s.getName(),"小天"))
        .findFirst()
        // orElse 表示如果 findFirst 返回 null 的话，就返回 orElse 里的内容
        .orElse(new StudentDTO()).getId();
    log.info("testFindFirst 小天同学的 ID {}",id2);

    Optional<StudentDTO> student= students.stream()
        .filter(s->StringUtils.equals(s.getName(),"小天"))
        .findFirst();
    // isPresent 为 true 的话，表示 value != null，即 student.get() != null
    if(student.isPresent()){
```


目录

```
log.info("testFindFirst 找不到名为小天的同学");  
}
```

运行结果如下:

```
@Test  
public void testFindFirst(){  
    Long id = students.stream()  
        .filter(s->StringUtils.equals(s.getName(),"小美"))  
        .findFirst()  
        .get().getId();  
  
    log.info("testFindFirst 小美同学的 ID {}",id);  
  
    // 防止空指针  
    Long id2 = students.stream()  
        .filter(s->StringUtils.equals(s.getName(),"小天"))  
        .findFirst()  
        // orElse 表示如果 findFirst 返回 null 的话, 就返回 orElse 里的内容  
        .orElse(new StudentDTO()).getId();  
    log.info("testFindFirst 小天同学的 ID {}",id2);  
  
    Optional<StudentDTO> student= students.stream()  
        .filter(s->StringUtils.equals(s.getName(),"小天"))  
        .findFirst();  
    // isPresent 为 true 的话, 表示 value != null  
    if(student.isPresent()){  
        log.info("testFindFirst 小天同学的 ID {}",student.get().getId());  
        return;  
    }  
    log.info("testFindFirst 找不到名为小天的同学");  
}
```

预防了可预见的未知错误

```
machines/jdk1.8.0_45.jdk/Contents/Home/bin/java ...  
demo.eight.LambdaExpressionDemo testFindFirst 小美同学的 ID 1  
demo.eight.LambdaExpressionDemo testFindFirst 小天同学的 ID null  
demo.eight.LambdaExpressionDemo testFindFirst 找不到名为小天的同学
```

2.11 groupingBy && toMap

groupingBy 是能够根据字段进行分组, toMap 是把 List 的数据格式转化成 Map 的格式, 我们写了一个 demo, 如下:

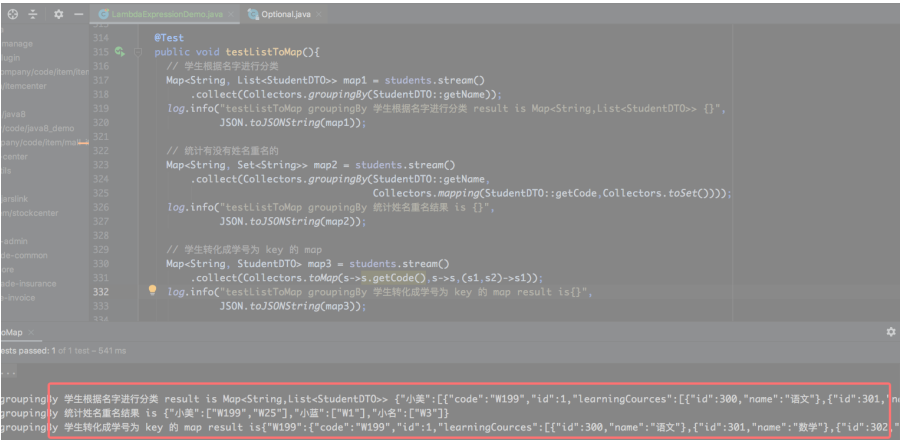
```
@Test  
public void testListToMap(){  
    // 学生根据名字进行分类  
    Map<String, List<StudentDTO>> map1 = students.stream()  
        .collect(Collectors.groupingBy(StudentDTO::getName));  
    log.info("testListToMap groupingBy 学生根据名字进行分类 result is Map<String,List<Student  
        JSON.toJSONString(map1));  
  
    // 统计姓名重名的学生有哪些  
    Map<String, Set<String>> map2 = students.stream()  
        .collect(Collectors.groupingBy(StudentDTO::getName,  
            Collectors.mapping(StudentDTO::getCode,Collectors.toSet())));  
    log.info("testListToMap groupingBy 统计姓名重名结果 is {}",  
        JSON.toJSONString(map2));  
  
    // 学生转化成学号为 key 的 map  
    Map<String, StudentDTO> map3 = students.stream()  
        //第一个入参表示 map 中 key 的取值
```

目录

```
log.info("testListToMap groupingBy 学生转化成学号为 key 的 map result is{}",
        JSON.toJSONString(map3));

}
```

运行结果如下：



3 总结

本文我们介绍了 12 种 Lambda 表达式常用的方法，大家可以找到 LambdaExpressionDemo 类，自己 debug 下，这样你在工作中遇到复杂数据结构转化时，肯定会得心应手了。

精选留言 1

欢迎在这里发表留言，作者筛选后可公开显示

weibo_可否争番一口气_0

不错，讲得比较详细，希望讲讲lambda的方法引用，一直不是很理解哪些方法或者场景可以用方法引用。

👍 0 回复

2019-12-04

文贺 回复 weibo_可否争番一口气_0

最常用方法引用的地方就是递归，我们写递归代码时，常常担心递归太深，造成栈溢出，或是不小心代码有漏洞，使本来预计只需递归几次的，递归了很多次，导致超出栈的深度，内存溢出，这时候就可以把需要递归的代码使用 Lambda 封装一下，这样不管递归多少层，内存都不会溢出。不推荐使用方法引用的地方就是远程 rpc 调用，我们常用的 dubbo 框架在处理入参和出参是方法引用时会丢失类型和值。

回复

7天前

目录