我是七月呀

JDK8-Stream流常用方法

Stream流的使用

流操作是Java8提供一个重要新特性，它允许开发人员以声明性方式处理集合，其核心类库主要改进了对集合类的 API和新增Stream操作。Stream类中每一个方法都对应集合上的一种操作。将真正的函数式编程引入到Java中，能 让代码更加简洁，极大地简化了集合的处理操作，提高了开发的效率和生产力。

同时stream不是一种数据结构，它只是某种数据源的一个视图，数据源可以是一个数组，Java容器或I/O channel等。在Stream中的操作每一次都会产生新的流，内部不会像普通集合操作一样立刻获取值，而是惰性 取值，只有等到用户真正需要结果的时候才会执行。**并且对于现在调用的方法，本身都是一种高层次构件，与线程模型无关。因此在并行使用中，开发者们无需再去操 心线程和锁了。Stream内部都已经做好了。**

如果刚接触流操作的话，可能会感觉不太舒服。其实理解流操作的话可以对比数据库操作。把流的操作理解为对数据库中 数据的查询操作

集合 = 数据表
元素 = 表中的每条数据
属性 = 每条数据的列
流API = sql查询

流操作详解

Stream流接口中定义了许多对于集合的操作方法，总的来说可以分为两大类：中间操作和终端操作。

- 中间操作：会返回一个流，通过这种方式可以将多个中间操作连接起来，形成一个调用链，从而转换为另外 一个流。除非调用链后存在一个终端操作，否则中间操作对流不会进行任何结果处理。
- 终端操作：会返回一个具体的结果，如boolean、list、integer等。

1、筛选

公告

昵称： 我是七月呀
园龄： 1年10个月
粉丝： 0
关注： 0
+加关注

< 2021年3月 >

日	一	二	三	四	五	六
28	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

搜索

找找看

谷歌搜索

随笔档案

2021年1月(2)

2020年12月(6)

阅读排行榜

对于集合的操作，经常性的会涉及到对于集中符合条件的数据筛选，Stream中对于数据筛选两个常见的API： filter(过滤)、distinct(去重)

1.1基于filter()实现数据过

该方法会接收一个返回boolean的函数作为参数，终返回一个包括所有符合条件元素的流。

案例：获取所有年龄20岁以下的学生

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class FilterDemo {
    public static void main(String[] args) {

        //获取所有年龄20岁以下的学生
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三","M",true));
        students.add(new Student(1,18,"李四","M",false));
        students.add(new Student(1,21,"王五","F",true));
        students.add(new Student(1,20,"赵六","F",false));
        students.stream().filter(student -> student.getAge()<20);
    }
}
```

源码解析

```
/**
 * Returns a stream consisting of the elements of this stream that match
 * the given predicate.
 *
 * <p>This is an <a href="package-summary.html#StreamOps">intermediate
 * operation</a>.
 *
 * @param predicate a <a href="package-summary.html#NonInterference">non-interfering</a>,
 *                  <a href="package-summary.html#Statelessness">stateless</a>
 *                  predicate to apply to each element to determine if it
 *                  should be included
 * @return the new stream
 */
Stream<T> filter(Predicate<? super T> predicate);
```

此处可以看到filter方法接收了Predicate函数式接口。

```
@Override
public final Stream<P_OUT> filter(Predicate<? super P_OUT> predicate) {
    Objects.requireNonNull(predicate);
    return new StatelessOp<P_OUT, P_OUT>( upstream: this, StreamShape.REFERENCE,
                                 StreamOpFlag.NOT_SIZED) {
        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<P_OUT> sink) {
            return new Sink.ChainedReference<P_OUT, P_OUT>(sink) {
                @Override
                public void begin(long size) { downstream.begin( size: -1); }

                @Override
                public void accept(P_OUT u) {
                    if (predicate.test(u))
                        downstream.accept(u);
                }
            };
        }
    };
}
```

首先判断predicate是否为null，如果为null，则抛出NullPointerException;构建Stream，重写opWrapsink方法。参数flags:下一个sink的标志位，供优化使用。参数sink：下一个sink，通过此参数将sink构造成单链。此时流已经构建好，但是因为begin（）先执行，此时是无法确定流中后续会存在多少元素的，所以传递-1，代表无法确定。最后调用Pridicate中的test，进行条件判断，将符合条件数据放入流中。

1.2基于distinct实现数据去重

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class DistinctDemo {
```

1. JDK8-Stream并行流详解(414)

2. JDK8-Stream流常用方法(368)

3. JDK8新特性详解（一）(324)

4. es6语法详解(101)

5. vagrant up报错【io.rb:32:in `encode': "\x95" followed by "\"" on GBK (Encoding::InvalidByteSequenceError)】(73)

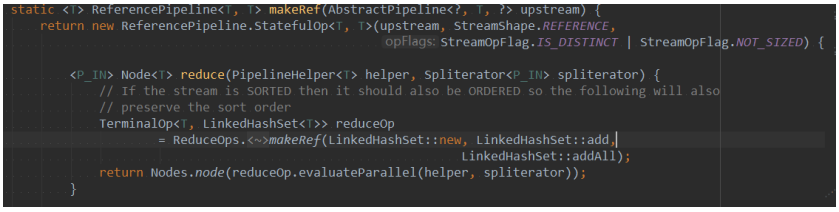
推荐排行榜

1. es6语法详解(1)

2. JDK8-Stream流常用方法(1)

```
    public static void main(String[] args) {

        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 4, 5, 5, 6,
7, 8, 2, 2, 2, 2);
        integers.stream().distinct().collect(Collectors.toList());

    }
}
```

源码解析



根据其源码，我们可以知道在distinct()内部是基于LinkedHashSet对流中数据进行去重，并终返回一个新的流。

2、切片

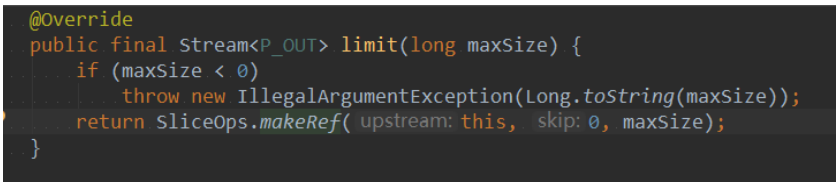
2.1基于limit()实现数据截取

该方法会返回一个不超过给定长度的流

案例：获取数组的前五位

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class LimitDemo {

    public static void main(String[] args) {
                //获取数组的前五位
        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 4, 5, 5, 6,
7, 8, 2, 2, 2, 2);
        integers.stream().limit(5);

    }
}
```

源码解析：



对于limit方法的实现，它会接收截取的长度，如果该值小于0，则抛出异常，否则会继续向下调用 SliceOps.makeRef()。该方法中this代表当前流，skip代表需要跳过元素，比方说本来应该有4个元素，当跳过元素 值为2，会跳过前面两个元素，获取后面两个。maxSize代表要截取的长度

```
/**
 * Appends a "slice" operation to the provided stream.  The slice operation
 * may be may be skip-only, limit-only, or skip-and-limit.
 *
 * @param <T> the type of both input and output elements
 * @param upstream a reference stream with element type T
 * @param skip the number of elements to skip.  Must be >= 0.
 * @param limit the maximum size of the resulting stream, or -1 if no limit
 *        is to be imposed
 */
public static <T> Stream<T> makeRef(AbstractPipeline<?, T, ?> upstream,
                                    long skip, long limit) {
    if (skip < 0)
        throw new IllegalArgumentException("Skip must be non-negative: " + skip);

    return new ReferencePipeline.StatefulOp<T, T>(upstream, StreamShape.REFERENCE,
                                                  flags(limit)) {
        Spliterator<T> unorderedSkipLimitSpliterator(Spliterator<T> s,
                                                     long skip, long limit, long sizeIfKnown) {
            if (skip <= sizeIfKnown) {
                // Use just the limit if the number of elements
                // to skip is <= the known pipeline size
                limit = limit >= 0 ? Math.min(limit, sizeIfKnown - skip) : sizeIfKnown - skip;
                skip = 0;
            }
            return new StreamSpliterators.UnorderedSliceSpliterator.OfRef<>(s, skip, limit);
        }
```

在makeRef方法中的unorderedSkipLimitSpliterator()中接收了四个参数Spliterator，skip(跳过个数)、limit(截取 个数)、sizeIfKnown(已知流大小)。如果跳过个数小于已知流大小，则判断跳过个数是否大于0，如果大于则取截取 个数或已知流大小-跳过个数的两者小值，否则取已知流大小-跳过个数的结果，作为跳过个数。后对集合基于跳过个数和截取个数进行切割。

2.2基于skip()实现数据跳过

案例：从集合第三个开始截取5个数据

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class LimitDemo {

    public static void main(String[] args) {
        //从集合第三个开始截取5个数据
        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 4, 5, 5, 6,
7, 8, 2, 2, 2, 2);
        List<Integer> collect =
integers.stream().skip(3).limit(5).collect(Collectors.toList());
        collect.forEach(integer -> System.out.print(integer+" "));

    }
}
```

```
结果4 4 5 5 6
```

案例：先从集合中截取5个元素，然后取后3个

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class LimitDemo {

    public static void main(String[] args) {
        //先从集合中截取5个元素，然后取后3个
        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 4, 5, 5, 6,
7, 8, 2, 2, 2, 2);
        List<Integer> collect =
integers.stream().limit(5).skip(2).collect(Collectors.toList());
        collect.forEach(integer -> System.out.print(integer+" "));

    }
}
```

```
结果: 3 4 4
```

源码分析：

```
@Override
public final Stream<P_OUT> skip(long n) {
    if (n < 0)
        throw new IllegalArgumentException(Long.toString(n));
    if (n == 0)
        return this;
    else
        return SliceOps.makeRef( upstream: this, n,  limit: -1);
}
```

在skip方法中接收的n代表的是要跳过的元素个数，如果n小于0，抛出非法参数异常，如果n等于0，则返回当前 流。如果n小于0，才会调用makeRef()。同时指定limit参数为-1.

```
 * @param <T> the type of both input and output elements
 * @param upstream a reference stream with element type T
 * @param skip the number of elements to skip.  Must be >= 0.
 * @param limit the maximum size of the resulting stream, or -1 if no limit
 *        is to be imposed
 */
public static <T> Stream<T> makeRef(AbstractPipeline<?, T, ?> upstream,
                                    long skip, long limit) {
    if (skip < 0)
        throw new IllegalArgumentException("Skip must be non-negative: " + skip);

    return new ReferencePipeline.StatefulOp<T, T>(upstream, StreamShape.REFERENCE,
                                                  flags(limit)) {
        Spliterator<T> unorderedSkipLimitSpliterator(Spliterator<T> s,
                                                     long skip, long limit, long sizeIfKnown) {
            if (skip <= sizeIfKnown) {
                // Use just the limit if the number of elements
                // to skip is <= the known pipeline size
                limit = limit >= 0 ? Math.min(limit, sizeIfKnown - skip) : sizeIfKnown - skip;
                skip = 0;
            }
```

此时可以发现limit和skip都会进入到该方法中，在确定limit值时，如果limit<0,则获取已知集合大小长度-跳过的长度。最终进行数据切割。

3、映射

在对集合进行操作的时候，我们经常会从某些对象中选择性的提取某些元素的值，就像编写sql一样，指定获取表 中特定的数据列

```
#指定获取特定列 SELECT name FROM student
```

在Stream API中也提供了类似的方法，map()。它接收一个函数作为方法参数，这个函数会被应用到集合中每一个 元素上，并终将其映射为一个新的元素。
案例：获取所有学生的姓名，并形成一个新的集合

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class MapDemo {
    public static void main(String[] args) {

        //获取所有学生的姓名，并形成一个新的集合
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三","M",true));
        students.add(new Student(1,18,"李四","M",false));
        students.add(new Student(1,21,"王五","F",true));
        students.add(new Student(1,20,"赵六","F",false));

        List<String> collect =
students.stream().map(Student::getName).collect(Collectors.toList());
        collect.forEach(s -> System.out.print(s + " "));
    }
}
```

```
结果：张三 李四 王五 赵六
```

源码解析：

内部对Function函数式接口中的apply方法进行实现，接收一个对象，返回另外一个对象，并把这个内容存入当前 流中，后返回

4、匹配

在日常开发中，有时还需要判断集合中某些元素是否匹配对应的条件，如果有的话，在进行后续的操作。在 Stream API中也提供了相关方法供我们进行使用，如anyMatch、allMatch等。他们对应的就是&&和||运算符。

4.1基于anyMatch()判断条件至少匹配一个元素

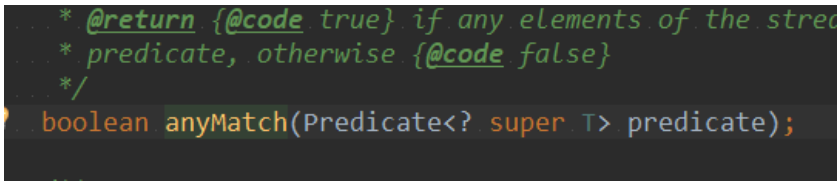
anyMatch()主要用于判断流中是否至少存在一个符合条件的元素，它会返回一个boolean值，并且对于它的操作， 一般叫做短路求值

案例：判断集合中是否有年龄小于20的学生

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class AnyMatchDemo {

    public static void main(String[] args) {
        //判断集合中是否有年龄小于20的学生
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三","M",true));
        students.add(new Student(1,18,"李四","M",false));
        students.add(new Student(1,21,"王五","F",true));
        students.add(new Student(1,20,"赵六","F",false));

        if(students.stream().anyMatch(student -> student.getAge() <
20)){
            System.out.println("集合中有年龄小于20的学生");
        }else {
            System.out.println("集合中没有年龄小于20的学生");
        }
    }
}
```

根据上述例子可以看到，当流中只要有一个符合条件的元素，则会立刻中止后续的操作，立即返回一个布尔值，无需遍历整个流。

源码解析：



内部实现会调用makeRef(),其接收一个Predicate函数式接口，并接收一个枚举值，该值代表当前操作执行的是 ANY。

```
enum MatchKind {
    /** Do all elements match the predicate? */
    ANY(true, true),

    /** Do any elements match the predicate? */
    ALL(false, false),

    /** Do no elements match the predicate? */
    NONE(true, false);

    private final boolean stopOnPredicateMatches;
    private final boolean shortCircuitResult;

    private MatchKind(boolean stopOnPredicateMatches,
                      boolean shortCircuitResult) {
        this.stopOnPredicateMatches = stopOnPredicateMatches;
        this.shortCircuitResult = shortCircuitResult;
    }
}
```

```
public static <T> TerminalOp<T, Boolean> makeRef(Predicate<? super T> predicate,
                                                 MatchKind matchKind) {
    Objects.requireNonNull(predicate);
    Objects.requireNonNull(matchKind);
    class MatchSink extends BooleanTerminalSink<T> {
        MatchSink() { super(matchKind); }

        @Override
        public void accept(T t) {
            if (!stop && predicate.test(t) == matchKind.stopOnPredicateMatches) {
                stop = true;
                value = matchKind.shortCircuitResult;
            }
        }
    }

    return new MatchOp<>(StreamShape.REFERENCE, matchKind, MatchSink::new);
}
```

如果test()抽象方法执行返回值==MatchKind中any的stopOnPredicateMatches，则将stop中断置为true，value 也为true。并终进行返回。无需进行后续的流操作。

4.2基于allMatch()判断条件是否匹配所有元素

allMatch()的工作原理与anyMatch()类似，但是anyMatch执行时，只要流中有一个元素符合条件就会返回true， 而allMatch会判断流中是否所有条件都符合条件，全部符合才会返回true

案例：判断集合所有学生的年龄是否都小于20

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class AllMatchDemo {

    public static void main(String[] args) {
        //判断集合所有学生的年龄是否都小于20
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三","M",true));
        students.add(new Student(1,18,"李四","M",false));
        students.add(new Student(1,21,"王五","F",true));
        students.add(new Student(1,20,"赵六","F",false));

        if(students.stream().allMatch(student -> student.getAge() <
20)){
            System.out.println("集合所有学生的年龄都小于20");
        }else {
            System.out.println("集合中有年龄大于20的学生");
        }
    }
}
```

源码解析：与anyMatch类似，只是其枚举参数的值为ALL

5、查找

对于集合操作，有时需要从集合中查找中符合条件的元素，Stream中也提供了相关的API，findAny()和 findFirst()，他俩可以与其他流操作组合使用。findAny用于获取流中随机的某一个元素，findFirst用于获取流中的 第一个元素。至于一些特别的定制化需求，则需要自行实现。

5.1基于findAny()查找元素

案例：findAny用于获取流中随机的某一个元素，并且利用短路在找到结果时，立即结束

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class FindAnyDemo {
    public static void main(String[] args) {
        //findAny用于获取流中随机的某一个元素，并且利用短路在找到结果时，立即结束
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三1","M",true));
        students.add(new Student(1,18,"张三2","M",false));
        students.add(new Student(1,21,"张三3","F",true));
        students.add(new Student(1,20,"张三4","F",false));
        students.add(new Student(1,20,"张三5","F",false));
        students.add(new Student(1,20,"张三6","F",false));
        Optional<Student> student1 = students.stream().filter(student -
> student.getSex().equals("F")).findAny();
        System.out.println(student1.toString());
    }

}
```

结果：`Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]`

此时我们将其循环100次

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class FindAnyDemo {
    public static void main(String[] args) {
        //findAny用于获取流中随机的某一个元素，并且利用短路在找到结果时，立即结束
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三1","M",true));
        students.add(new Student(1,18,"张三2","M",false));
        students.add(new Student(1,21,"张三3","F",true));
        students.add(new Student(1,20,"张三4","F",false));
        students.add(new Student(1,20,"张三5","F",false));
        students.add(new Student(1,20,"张三6","F",false));
        for (int i = 0; i < 100; i++) {
            Optional<Student> student1 =
students.stream().filter(student ->
student.getSex().equals("F")).findAny();
            System.out.println(student1.toString());
        }

    }

}
```

结果：

```
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
```

由于数量较大，只截取了部分截图，全部都是一样的，不行的小伙伴可以自己测试一下

这时候我们改为串行流在执行一下

```java
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class FindAnyDemo {
    public static void main(String[] args) {
        //findAny用于获取流中随机的某一个元素，并且利用短路在找到结果时，立即结束
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三1","M",true));
        students.add(new Student(1,18,"张三2","M",false));
        students.add(new Student(1,21,"张三3","F",true));
        students.add(new Student(1,20,"张三4","F",false));
        students.add(new Student(1,20,"张三5","F",false));
        students.add(new Student(1,20,"张三6","F",false));
        for (int i = 0; i < 100; i++) {
            Optional<Student> student1 =
students.parallelStream().filter(student ->
student.getSex().equals("F")).findAny();
            System.out.println(student1.toString());
        }

    }

}
```

结果：

```
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=20, name='张三4', sex='F', isPass=false}]
Optional[Student{id=1, age=21, name='张三3', sex='F', isPass=true}]
```

现在我们通过源码解析来分析下这是为什么？

```
/**
 * Returns an {@link Optional} describing some element of the stream, or an
 * empty {@code Optional} if the stream is empty.
 *
 * <p>This is a <a href="package-summary.html#StreamOps">short-circuiting
 * terminal operation</a>.
 *
 * <p>The behavior of this operation is explicitly nondeterministic; it is
 * free to select any element in the stream.  This is to allow for maximal
 * performance in parallel operations; the cost is that multiple invocations
 * on the same source may not return the same result.  (If a stable result
 * is desired, use {@link #findFirst()} instead.)
 *
 * @return an {@code Optional} describing some element of this stream, or an
 * empty {@code Optional} if the stream is empty
 * @throws NullPointerException if the element selected is null
 * @see #findFirst()
 */
Optional<T> findAny();
```

根据这一段源码介绍，findAny对于同一数据源的多次操作会返回不同的结果。但是，我们现在的操作是串行的， 所以在数据较少的情况下，一般会返回第一个结果，但是如果在并行的情况下，那就不能确保返回的是第一个了。 这种设计主要是为了获取更加高效的性能。并行操作后续会做详细介绍。

```
@Override
public final Optional<P_OUT> findAny() {
    return evaluate(FindOps.makeRef( mustFindFirst: false));
}
```

传递参数，指定不必须获取第一个元素

```
*/
final <R> R evaluate(TerminalOp<E_OUT, R> terminalOp) {
    assert getOutputShape() == terminalOp.inputShape();
    if (linkedOrConsumed)
        throw new IllegalStateException(MSG_STREAM_LINKED);
    linkedOrConsumed = true;

    return isParallel()
           ? terminalOp.evaluateParallel( helper: this, sourceSpliterator(terminalOp.getOpFlags()))
           : terminalOp.evaluateSequential( helper: this, sourceSpliterator(terminalOp.getOpFlags()));
}
```

在该方法中，主要用于判断对于当前的操作执行并行还是串行。

```
@Override
public <S> O evaluateSequential(PipelineHelper<T> helper,
                                Spliterator<S> spliterator) {
    O result = helper.wrapAndCopyInto(sinkSupplier.get(), spliterator).get();
    return result != null ? result : emptyValue;
}
```

在该方法中的wrapAndCopyInto()内部做的会判断流中是否存在符合条件的元素，如果有的话，则会进行返回。结 果终会封装到Optional中的IsPresent中。

总结：当为串行流且数据较少时，获取的结果一般为流中第一个元素，但是当为并流行的时 候，则会随机获取。

5.2基于findFirst()查找元素

findFirst使用原理与findAny类似，只是它无论串行流还是并行流都会返回第一个元素，这里不做详解

6、归约

到现在截止，对于流的终端操作，我们返回的有boolean、Optional和List。但是在集合操作中，我们经常会涉及 对元素进行统计计算之类的操作，如求和、求大值、小值等，从而返回不同的数据结果。

6.1基于reduce()进行累积求和

案例：对集合中的元素求和

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
```

```
 */
public class ReduceDemo {
    public static void main(String[] args) {
        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 4, 5, 5, 6,
7, 8, 2, 2, 2, 2);
        Integer reduce = integers.stream().reduce(0, (integer1,
integer2) -> integer1 + integer2);
        System.out.println(reduce);
    }
}
```

结果：53

在上述代码中，在reduce里的第一个参数声明为初始值，第二个参数接收一个lambda表达式，代表当前流中的两 个元素，它会反复相加每一个元素，直到流被归约成一个终结果

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

优化成这样也是可以的。当然，reduce还有一个不带初始值参数的重载方法，但是要对返回结果进行判断，因为如果流中没有任何元素的话，可能就没有结果了。具体方法如下所示

```
   List<Integer> integers = Arrays.asList(1, 2, 3, 4, 4, 5, 5, 6, 7, 8,
2, 2, 2, 2);
        Optional<Integer> reduce =
integers.stream().reduce(Integer::sum);
        if(reduce.isPresent()){
            System.out.println(reduce);
        }else {
            System.out.println("数据有误");
        }
```

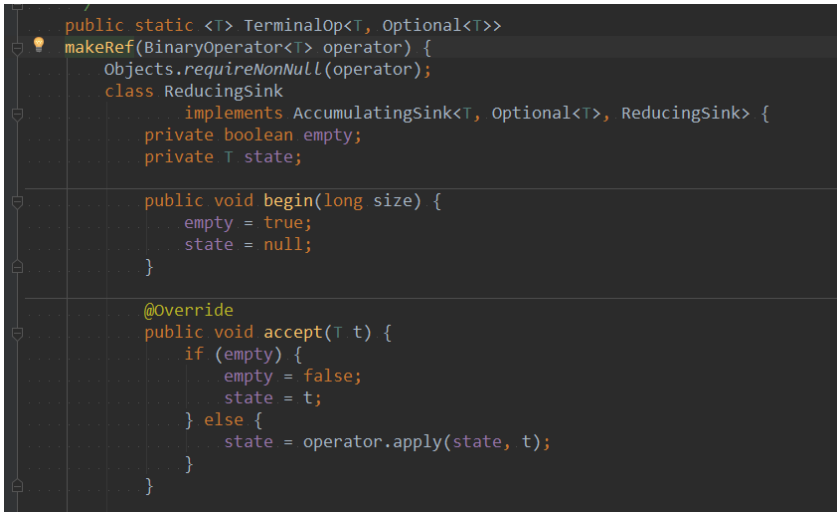
源码解析：两个参数的reduce方法

```
public static <T, U> TerminalOp<T, U>
makeRef(U seed, BiFunction<U, ? super T, U> reducer, BinaryOperator<U> combiner) {
    Objects.requireNonNull(reducer);
    Objects.requireNonNull(combiner);
    class ReducingSink extends Box<U> implements AccumulatingSink<T, U, ReducingSink> {
        @Override
        public void begin(long size) { state = seed; }

        @Override
        public void accept(T t) { state = reducer.apply(state, t); }

        @Override
        public void combine(ReducingSink other) { state = combiner.apply(state, other.state); }
    }
    return new ReduceOp<T, U, ReducingSink>(StreamShape.REFERENCE) {
        @Override
        public ReducingSink makeSink() { return new ReducingSink(); }
    };
}
```

在上述方法中，对于流中元素的操作，当执行第一个元素，会进入begin方法，将初始化的值给到state，state就 是后的返回结果。并执行accept方法，对state和第一个元素根据传入的操作，对两个值进行计算。并把终计 算结果赋给state。
当执行到流中第二个元素，直接执行accept方法，对state和第二个元素对两个值进行计算，并把终计算结果赋 给state。后续依次类推。
可以按照下述代码进行理解

```
T result = identity;
for (T element : this stream){
result = accumulator.apply(result, element)
}
return result;
```

源码解析：单个参数的reduce方法



```
public static <T> TerminalOp<T, Optional<T>>
makeRef(BinaryOperator<T> operator) {
    Objects.requireNonNull(operator);
    class ReducingSink
            implements AccumulatingSink<T, Optional<T>, ReducingSink> {
        private boolean empty;
        private T state;

        public void begin(long size) {
            empty = true;
            state = null;
        }

        @Override
        public void accept(T t) {
            if (empty) {
                empty = false;
                state = t;
            } else {
                state = operator.apply(state, t);
            }
        }
```

在这部分实现中，对于匿名内部类中的empty相当于是一个开关，state相当于结果。

对于流中第一个元素，首先会执行begin()将empty置为true，state为null。接着进入到accept()，判断empty是否 为true，如果为true，则将empty置为false，同时state置为当前流中第一个元素，当执行到流中第二个元素时， 直接进入到accpet()，判断empty是否为true，此时empty为false，则会执行apply()，对当前state和第二个元素进 行计算，并将结果赋给state。后续依次类推。
当整个流操作完之后，执行get(), 如果empty为true，则返回一个空的Optional对象，如果为false，则将后计算 完的state存入Optional中。

可以按照下述代码进行理解：

```
boolean flag = false;
T result = null;
for (T element : this stream) {
    if (!flag) {
        flag = true;
        result = element;
    }else{
        result = accumulator.apply(result, element);
    }
}
return flag ? Optional.of(result) : Optional.empty();
```

6.2获取流中元素的最大值、最小值

案例：获取集合中元素的最大值、最小值

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class MaxDemo {
    public static void main(String[] args) {

        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 4, 5, 5, 6,
7, 8, 2, 2, 2, 2);

        /**
         * 获取集合中的最大值
         */
        //方法一
        Optional<Integer> max1 =
integers.stream().reduce(Integer::max);
        if(max1.isPresent()){
            System.out.println(max1);
        }
        //方法二
        Optional<Integer> max2 =
integers.stream().max(Integer::compareTo);
```

```
        if(max2.isPresent()){
            System.out.println(max2);
        }

        /**
         * 获取集合中的最小值
         */
        //方法一
        Optional<Integer> min1 =
integers.stream().reduce(Integer::min);
        if(min1.isPresent()){
            System.out.println(min1);
        }

        //方法二
        Optional<Integer> min2 =
integers.stream().min(Integer::compareTo);
        if(min2.isPresent()){
            System.out.println(min2);
        }
    }
}
```

结果：

```
Optional[8]
Optional[8]
Optional[1]
Optional[1]
```

7、收集器

通过使用收集器，可以让代码更加方便的进行简化与重用。其内部主要核心是通过Collectors完成更加复杂的计算 转换，从而获取到终结果。并且Collectors内部提供了非常多的常用静态方法，直接拿来就可以了。比方说： toList。

```
/**
 * @author 我是七月呀
 * @date 2020/12/22
 */
public class CollectDemo {
    public static void main(String[] args) {
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(1,19,"张三","M",true));
        students.add(new Student(1,18,"李四","M",false));
        students.add(new Student(1,21,"王五","F",true));
        students.add(new Student(1,20,"赵六","F",false));

        //通过counting()统计集合总数  方法一
        Long collect =
students.stream().collect(Collectors.counting());
        System.out.println(collect);
        //结果 4

        //通过count()统计集合总数  方法二
        long count = students.stream().count();
        System.out.println(count);
        //结果 4

        //通过maxBy求最大值
        Optional<Student> collect1 =
students.stream().collect(Collectors.maxBy(Comparator.comparing(Student
::getAge)));
        if(collect1.isPresent()){
            System.out.println(collect1);
        }
        //结果 Optional[Student{id=1, age=21, name='王五', sex='F',
isPass=true}]

        //通过max求最大值
        Optional<Student> max =
students.stream().max(Comparator.comparing(Student::getAge));
```

```
        if(max.isPresent()){
            System.out.println(max);
        }
        //结果  Optional[Student{id=1, age=21, name='王五', sex='F',
isPass=true}]

        //通过minBy求最小值
        Optional<Student> collect2 =
students.stream().collect(Collectors.minBy(Comparator.comparing(Student
::getAge)));
        if(collect2.isPresent()){
            System.out.println(collect2);
        }
        //结果  Optional[Student{id=1, age=18, name='李四', sex='M',
isPass=false}]

        //通过min求最小值
        Optional<Student> min =
students.stream().min(Comparator.comparing(Student::getAge));
        if(min.isPresent()){
            System.out.println(min);
        }
        //结果  Optional[Student{id=1, age=18, name='李四', sex='M',
isPass=false}]

        //通过summingInt()进行数据汇总
        Integer collect3 =
students.stream().collect(Collectors.summingInt(Student::getAge));
        System.out.println(collect3);
        //结果 78

        //通过averagingInt()进行平均值获取
        Double collect4 =
students.stream().collect(Collectors.averagingInt(Student::getAge));
        System.out.println(collect4);
        //结果 19.5

        //通过joining()进行数据拼接
        String collect5 =
students.stream().map(Student::getName).collect(Collectors.joining());
        System.out.println(collect5);
        //结果 张三李四王五赵六

        //复杂结果的返回
        IntSummaryStatistics collect6 =
students.stream().collect(Collectors.summarizingInt(Student::getAge));
        double average = collect6.getAverage();
        long sum = collect6.getSum();
        long count1 = collect6.getCount();
        int max1 = collect6.getMax();
        int min1 = collect6.getMin();

    }

}
```

8、分组

在数据库操作中，经常会通过group by对查询结果进行分组。同时在日常开发中，也经常会涉及到这一类操作， 如通过性别对学生集合进行分组。如果通过普通编码的方式需要编写大量代码且可读性不好。

对于这个问题的解决，java8也提供了简化书写的方式。通过 Collectors。groupingBy()即可。

```
//通过性别对学生进行分组
        Map<String, List<Student>> collect =
students.stream().collect(Collectors.groupingBy(Student::getSex));
```

```
结果  {
    F=[Student{id=1, age=21, name='王五', sex='F', isPass=true},
Student{id=1, age=20, name='赵六', sex='F', isPass=false}],
```

```
    M=[Student{id=1, age=19, name='张三', sex='M', isPass=true},
Student{id=1, age=18, name='李四', sex='M', isPass=false}]
}
```

8.1多级分组

刚才已经使用groupingBy()完成了分组操作，但是只是通过单一的sex进行分组，那现在如果需求发生改变，还要 按照是否及格进行分组，能否实现？答案是可以的。对于groupingBy()它提供了两个参数的重载方法，用于完成这 种需求。

这个重载方法在接收普通函数之外，还会再接收一个Collector类型的参数，其会在内层分组(第二个参数)结果，传 递给外层分组(第一个参数)作为其继续分组的依据。

```
//现根据是否通过考试对学生分组，在根据性别分组
Map<String, Map<Boolean, List<Student>>> collect1 =
students.stream().collect(Collectors.groupingBy(Student::getSex,
Collectors.groupingBy(Student::getPass)));
```

```
结果： {
    F={
        false=[Student{id=1, age=20, name='赵六', sex='F',
isPass=false}],
        true=[Student{id=1, age=21, name='王五', sex='F', isPass=true}]
    },
    M={
        false=[Student{id=1, age=18, name='李四', sex='M',
isPass=false}],
        true=[Student{id=1, age=19, name='张三', sex='M', isPass=true}]}
}
```

8.2多级分组变形

在日常开发中，我们很有可能不是需要返回一个数据集合，还有可能对数据进行汇总操作，比方说对于年龄18岁 的通过的有多少人，未及格的有多少人。因此，对于二级分组收集器传递给外层分组收集器的可以任意数据类型， 而不一定是它的数据集合。

```
//根据年龄进行分组，获取并汇总人数
        Map<Integer, Long> collect2 =
students.stream().collect(Collectors.groupingBy(Student::getAge,
Collectors.counting()));
        System.out.println(collect2);
```

```
结果：{18=1, 19=1, 20=1, 21=1}
```

```
//要根据年龄与是否及格进行分组，并获取每组中年龄的学生
        Map<Integer, Map<Boolean, Student>> collect3 =
students.stream().collect(Collectors.groupingBy(Student::getAge,
Collectors.groupingBy(Student::getPass,

Collectors.collectingAndThen(Collectors.maxBy(Comparator.comparing(Stud
ent::getAge)), Optional::get))));
        System.out.println(collect3.toString());
```

```
结果：{
    18={false=Student{id=1, age=18, name='李四', sex='M',
isPass=false}},
    19={true=Student{id=1, age=19, name='张三', sex='M', isPass=true}},
    20={false=Student{id=1, age=20, name='赵六', sex='F',
isPass=false}},
    21={true=Student{id=1, age=21, name='王五', sex='F', isPass=true}}}
```


我是七月呀
关注 - 0
粉丝 - 0

+加关注

« 上一篇： JDK8新特性详解（一）

» 下一篇： JDK8-Stream并行流详解

posted @ 2020-12-23 16:45 我是七月呀 阅读(368) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

登录后才能发表评论，立即 登录 或 注册， 访问 网站首页