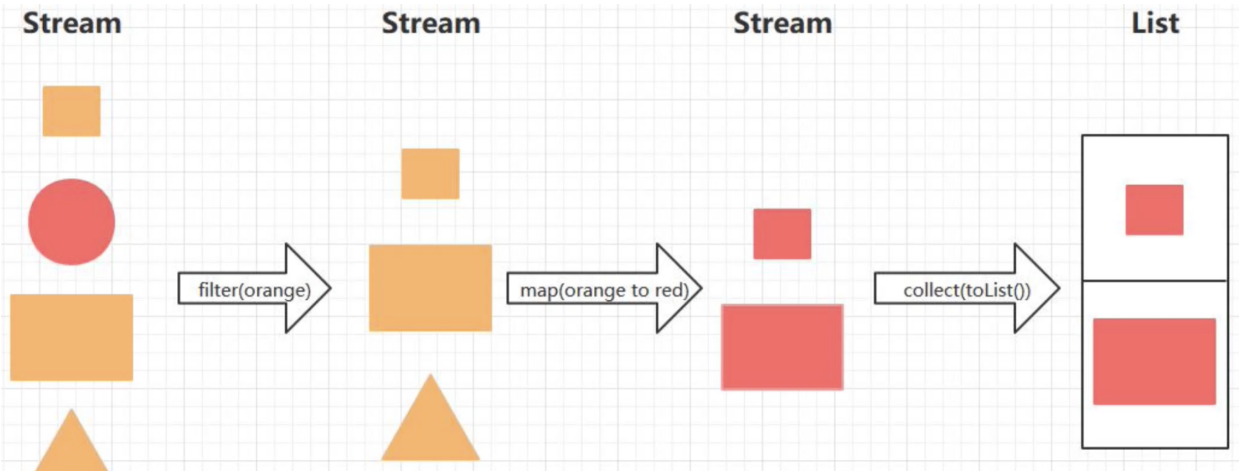
📖 青空的B站课程 原文链接：https://www.itbaima.cn/document/erpm32wduoaaqmrx

📅 2024年4月19日

> ps: 这节课我没有做很多的深入理解，直接拿青空的笔记然后自己的代码敲一下

Stream 流

⭐ Java 8 API添加了一个新的抽象称为流`Stream`，可以让你以一种声明的方式处理数据。Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 `Java` 集合运算和表达的高阶抽象。`Stream API`可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。这种风格将要处理的元素集合看作一种流， 流在管道中传输， 并且可以在管道的节点上进行处理， 比如筛选， 排序，聚合等。元素流在管道中经过中间操作（`intermediate operation`）的处理，最后由最终操作(`terminal operation`)得到前面处理的结果



⭐ 下面是一个代码实列

1️⃣ 这里是通过一个简单的应用来解释

```
    public static void main(String[] args) {
        List<String> list = new ArrayList<>
(Arrays.asList("AAA","b","CccC","dDdd", "AAA"));
        //查询字符长度大于3的字符串
        //查询以小写字母开头的字符串
        //过滤掉重复元素
        list = list.stream() //获取流
                .filter(s -> s.length() > 3) //过滤掉不符合要求的
                .filter(s -> s.charAt(0) >= 'a' && s.charAt(0) <= 'z') //同理
                .distinct() //过滤掉重复元素
                .collect(Collectors.toList()); //将进过流水线加工的元素重新收集起来，
通过Collections工具的toList方法转换为list
        System.out.println(list);
    }
```

2️⃣ `filter`内部传入的是一个断言函数式接口(具体可以看看泛型笔记中的函数式接口)

```
Stream<T> filter(Predicate<? super T> predicate); //在Stream接口中的定义
```

3 `collect` 内传入的是一个 `Collection`

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

⭐ 第二个实列

```
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>(Arrays.asList(1,4,6,7,9,0));
        //将List中的元素进行排列
        //将大于5的元素+1
        //限制两个元素
        list = list.stream()
                .sorted() //排序
                .map(str -> {if(str > 5) return str+1;return str;}) //自定义元素加工
                .limit(2) //限制元素大小
                .collect(Collectors.toList());
        System.out.println(list);
    }
```

1 可以先看看 `sroted`,同样也是闯入一个个比较器

```
    @Override
    public final Stream<P_OUT> sorted() {
        return SortedOps.makeRef(this);
    }

    @Override
    public final Stream<P_OUT> sorted(Comparator<? super P_OUT> comparator) {
        return SortedOps.makeRef(this, comparator);
    }
```

2 `map`

```
    @Override
    @SuppressWarnings("unchecked")
    public final <R> Stream<R> map(Function<? super P_OUT, ? extends R> mapper) {
        Objects.requireNonNull(mapper);
        return new StatelessOp<P_OUT, R>(this, StreamShape.REFERENCE,
                                     StreamOpFlag.NOT_SORTED | StreamOpFlag.NOT_DISTINCT) {
            @Override
            Sink<P_OUT> opWrapSink(int flags, Sink<R> sink) {
                return new Sink.ChainedReference<P_OUT, R>(sink) {
                    @Override
                    public void accept(P_OUT u) {
                        downstream.accept(mapper.apply(u));
                    }
                };
```

```
            }
        };
    }
```

3 `limit`

```
@Override
public final Stream<P_OUT> limit(long maxSize) {
    if (maxSize < 0)
        throw new IllegalArgumentException(Long.toString(maxSize));
    return SliceOps.makeRef(this, 0, maxSize);
}
```

⭐ 但是它的操作并不是一步一步执行的，而是等到`collect`收集才去执行

⭐ 如果是要通过`collect`来收集一个`LinkedList`需要使用`toCollection`,使用方法引用调用`new`方法

```
.collect(Collectors.toCollection(LinkedList::new));
```

⭐ `Radom`的`stream`

```
public static void main(String[] args) {
    Random random = new Random();  //没想到吧，Random支持直接生成随机数的流
    random
            .ints(-100, 100)   //生成-100~100之间的，随机int型数字（本质上是一个IntStream）
            .limit(10)   //只获取前10个数字（这是一个无限制的流，如果不加以限制，将会无限进行下去！）
            .filter(i -> i < 0)   //只保留小于0的数字
            .sorted()    //默认从小到大排序
            .forEach(System.out::println);   //依次打印
}
```

```
public static void main(String[] args) {
    Random random = new Random();  //Random是一个随机数工具类
    IntSummaryStatistics statistics = random
            .ints(0, 100)
            .limit(100)
            .summaryStatistics();    //获取语法统计实例
    System.out.println(statistics.getMax());  //快速获取最大值
    System.out.println(statistics.getCount());  //获取数量
    System.out.println(statistics.getAverage());   //获取平均值
}
```

⭐ 普通`List`

```
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        IntSummaryStatistics intSummaryStatistics= list.stream()
                .mapToInt(i -> i)    //将每一个元素映射为Integer类型（这里因为本来就是Integer）
                .summaryStatistics();
        System.out.println(intSummaryStatistics.getMax());
    }
```

我们还可以通过`flat`来对整个流进行进一步细分：

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("A,B");
    list.add("C,D");
    list.add("E,F");   //我们想让每一个元素通过,进行分割，变成独立的6个元素
    list = list
            .stream()    //生成流
            .flatMap(e -> Arrays.stream(e.split(",")))    //分割字符串并生成新的流
            .collect(Collectors.toList());   //汇成新的List
    System.out.println(list);   //得到结果
}
```

我们也可以只通过Stream来完成所有数字的和，使用`reduce`方法：

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(2);
    list.add(3);
    int sum = list
            .stream()
            .reduce((a, b) -> a + b)   //计算规则为：a是上一次计算的值，b是当前要计算的参数，这里是求和
            .get();    //我们发现得到的是一个Optional类实例，通过get方法返回得到的值
    System.out.println(sum);
}
```