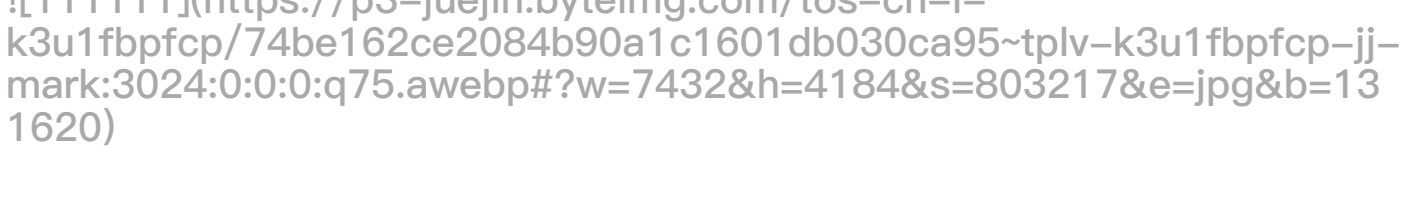
Java学习十一——Java8特性之Stream流

一、Java8新特性简介

> 2014年3月18日，JDK8发布，提供了Lambda表达式支持、内置Nashorn JavaScript引擎支持、新的时间日期API、彻底移除HotSpot永久代。



Java 8引入了许多令人兴奋的新特性，其中最引人注目的是Lambda表达式和Stream API。以下是Java 8的一些主要新特性介绍：

0. Lambda表达式：Lambda表达式是Java 8中最重要的特性之一。它允许您以更简洁的方式编写匿名函数，并将其作为参数传递给方法。Lambda表达式使代码更易读和更易维护。
1. Stream API：Stream API为集合框架新增了对函数式编程的支持。通过Stream API，您可以以声明性的方式处理集合数据，例如过滤、映射、排序等操作。这样可以更容易地编写并发、并行化的代码。
2. 接口的默认方法：Java 8允许在接口中定义默认方法。这使得接口可以包含具体的方法实现，而不仅仅是抽象方法。这样可以在不破坏现有实现的情况下向接口添加新的方法。
3. 方法引用：方法引用是一种更简洁的Lambda表达式的替代方式。它允许您直接引用现有方法或构造函数，而不必重新编写Lambda表达式。
4. 新的时间日期API：Java 8引入了全新的时间日期API，即java.time包。这个API解决了旧的Date和Calendar类存在的许多问题，并提供了更好的日期和时间处理功能。
5. CompletableFuture：CompletableFuture是一种新的异步编程工具，用于简化异步任务的处理。它提供了更灵活的方式来处理异步操作的结果和异常。
6. Nashorn JavaScript引擎：Java 8引入了Nashorn JavaScript引擎，用于在Java应用程序中执行JavaScript代码。这使得Java与JavaScript之间的互操作更加方便。
7. Optional 类：`Optional<T>` 类用于表示可能为空的值，避免了显式使用 `null`，减少了空指针异常的风险。

二、Stream流

2.1关于Stream流

2.1.1简介

Java Stream API 是 Java 8 引入的一项强大功能，主要用于处理集合数据。它提供了一种声明性的方法来处理数据，使代码更加简洁和易读。

2.1.2特点

Stream API 的一些关键特点：

0. **声明式编程**：使用 Stream API 时，你只需要指定“做什么”，而不需要关心“怎么做”。这使得代码更加简洁、易读。
1. **惰性求值**：Stream API 的操作是惰性的，这意味着在调用终端操作之前，中间操作不会执行。这有助于提高性能，因为它允许 JVM 优化操作的执行。
2. **不可变性**：Stream 本身是不可变的，一旦创建，就不能修改。这有助于避免并发修改异常。
3. **并行处理**：Stream API 支持并行处理，可以利用多核处理器提高性能。只需将 Stream 转换为并行 Stream，就可以并行执行操作。
4. **丰富的操作**：Stream API 提供了丰富的中间操作（如 `filter`、`map`、`reduce`）和终端操作（如 `forEach`、`collect`、`min`、`max`）。
5. **类型安全**：Stream API 与 Java 的泛型系统紧密集成，确保了类型安全

。
6. **无状态与有状态操作**：中间操作分为无状态操作（如 `filter`）和有状态操作（如 `distinct`、`sorted`）。无状态操作的结果只依赖于当前元素，而有状态操作可能需要考虑多个元素。
7. **短路操作**：某些终端操作（如 `anyMatch`、`allMatch`、`noneMatch`）可以在满足特定条件时提前终止，这称为短路操作。
8. **Optional 支持**：某些操作（如 `findFirst`、`findAny`）返回 `Optional` 类型，以避免空指针异常。
9. **集合操作**：Stream API 可以轻松地与集合框架集成，如使用 `Collection.stream()` 方法将集合转换为 Stream。

2.1.3使用流程

Java Stream 的一般流程：创建流 → 应用中间操作（可以有多个中间操作）→ 应用终端操作。流的操作可以链式调用，形成流畅的操作序列，提高了代码的简洁性和可读性。同时，Java Stream 也利用了并行处理来提升性能，特别是对于大数据集合的处理。

2.2创建stream流

* 从集合创建：

```
List<String> list = Arrays.asList(”a”, ”b”, ”c”);
Stream<String> streamFromList = list.stream();
```

* 从数组创建：

```
String[] array = {”a”, ”b”, ”c”};
Stream<String> streamFromArray = Arrays.stream(array);
```

```
* 使用 Stream.of() 方法：


```
Stream<String> stream = Stream.of(”a”, ”b”, ”c”);
```
* 从文件创建（使用 NIO）：


```
try (Stream<String> lines = Files.lines(Paths.get(”file.txt”))) {
    // 使用 lines 进行操作
} catch (IOException e) {
    e.printStackTrace();
}
```


2.2.1list集合创建


```
List<String> strings = Arrays.asList(”abc”, ””, ”bc”, ”efg”, ”abcd”,””, ”jkl”);
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
```


2.2.2map集合创建

keyset

那么这里的keyset当然就是把双列集合里的所有的键数据的set集合放在流里面
```

了

```
//    双列集合
        HashMap<String,Integer> map = new HashMap<>();
        map.put(”zhangsan”,23);
        map.put(”lisi”,24);
        map.put(”wangwu”,25);
        map.keySet().stream().forEach(s -> System.out.println(s));
```

entryset

```
//    双列集合
        HashMap<String,Integer> map = new HashMap<>();
        map.put(”zhangsan”,23);
        map.put(”lisi”,24);
        map.put(”wangwu”,25);
//       map.keySet().stream().forEach(s -> System.out.println(s));
        map.entrySet().stream().forEach(s-> System.out.println(s));
```

2.2.3数组创建流

可以使用数组的帮助类Arrays中的静态方法stream生成流

```
//        数组
        int  [] arr ={1,2,3,4,5};
        Arrays.stream(arr).forEach(s-> System.out.println(s));
```

2.2.4同种数据类型的多个数据

可以通过Stream里面的of方法，来获取到一个stream流
Stream.of（T...Values）生成流，可以看到这个of方法里是一个可变参数，所以不管传多少参数都是可以的

```
//同种数据类型的多个数据
    Stream.of(1,2,3,4,5).forEach(s-> System.out.println(s));
```

2.2.5并行流-Parallel-Streams

前面章节我们说过，`stream` 流是支持**顺序**和**并行**的。顺序流操作是单线程操作，而并行流是通过多线程来处理的，能够充分利用物理机 多核 CPU 的优势，同时处理速度更快。

测试：

首先，我们创建一个包含 1000000 UUID list 集合。

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
    UUID uuid = UUID.randomUUID();
    values.add(uuid.toString());
}
```

```

分别通过顺序流和并行流，对这个 list 进行排序，测算耗时:

##### 顺序流排序

```
// 纳秒
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

// 纳秒转微秒
long millis = TimeUnit.NANOSECONDS.toMillis(t1 – t0);
System.out.println(String.format("顺序流排序耗时: %d ms", millis));

// 顺序流排序耗时: 899 ms
```

##### 并行流排序

```
// 纳秒
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

// 纳秒转微秒
long millis = TimeUnit.NANOSECONDS.toMillis(t1 – t0);
System.out.println(String.format("并行流排序耗时: %d ms", millis));
```
```

// 并行流排序耗时: 472 ms

```

正如你所见，同样的逻辑处理，通过并行流，我们的性能提升了近 **50%** 。完成这一切，我们需要做的仅仅是将 `stream` 改成了 `parallelStream`。

#### 区别

`stream()` 和 `Parallel-Streams` 是 Java 8 引入的流式操作 API 中的两个重要概念。它们都属于 Java 8 新增的 Stream API，用于处理集合数据的函数式编程方式。

`stream()` 方法返回一个顺序流（sequential stream），它将集合转换为一个按顺序处理的流。顺序流适用于串行处理数据的场景，即每个元素依次经过一系列的中间操作和终端操作。

`Parallel-Streams` 则返回一个并行流（parallel stream），它将集合转换为一个可以并行处理的流。并行流适用于需要并行处理大量数据的场景，可以充分利用多核处理器的优势，提高处理速度。

使用场景区别如下：

0. `stream()`：适用于处理规模较小的数据集或要求顺序处理的场景。顺序流的处理过程是串行的，适合对数据进行逐个处理，保持处理顺序的情况。
1. `Parallel-Streams`：适用于处理大规模数据集或可以并行处理的场景。并行流的处理过程可以同时使用多个线程进行处理，能够加速处理速度。但需要注意，并行流的处理可能会引入==线程安全==问题，需要谨慎处理共享状态。

需要注意的是，并行流的性能提升并不是适用于所有场景的，有时候并行处理的开销可能会超过性能收益。因此，在选择使用 `Parallel-Streams` 时，需要根据具体情况进行评估和测试。
```

综上所述，`stream()` 适用于顺序处理小规模数据集的场景，而 `Parallel-Streams` 适用于并行处理大规模数据集的场景，可以根据实际需求选择合适的流类型。

2.3中间操作

Java Stream API 的中间操作是一系列操作，它们创建了一个新的 Stream，并且可以在这个新的 Stream 上继续添加操作。中间操作不会立即执行，它们是惰性的，只有在终端操作被调用时才会实际执行。以下是一些常见的中间操作方法及其示例：

2.3.1filter 过滤

筛选出符合条件的元素。

示例

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream()
                                   .filter(n -> n % 2 == 0)
                                   .collect(Collectors.toList());
// 输出: [2, 4]
```

过滤元素以某一字母开头

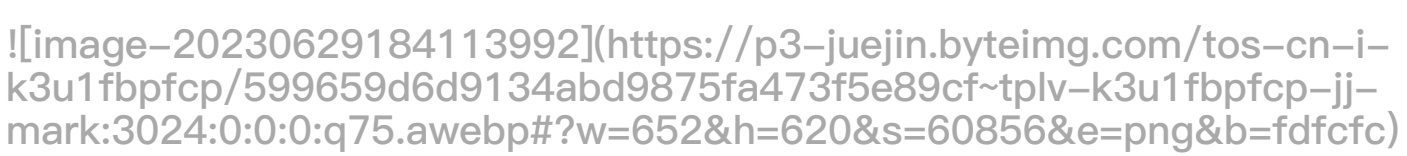
首先，我们创建一个 `List` 集合：

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add(”ddd2”);
stringCollection.add(”aaa2”);
stringCollection.add(”bbb1”);
stringCollection.add(”aaa1”);
stringCollection.add(”bbb3”);
stringCollection.add(”ccc”);
stringCollection.add(”bbb2”);
stringCollection.add(”ddd1”);

```

`Filter` 的入参是一个 `Predicate`，上面已经说到，`Predicate` 是一个断言的中间操作，它能够帮我们筛选出我们需要的集合元素。它的返参同样 是一个 `Stream` 流，我们可以通过 `foreach` 终端操作，来打印被筛选的元素：

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith(”a”))
    .forEach(System.out::println);

// ”aaa2”, ”aaa1”

```

> 注意：`foreach` 是一个终端操作，它的返参是 `void`，我们无法对其再次进行流操作。



2.3.2map 转换

将流中的元素通过指定的映射函数转换成另一个值。

示例

```
List<String> words = Arrays.asList("hello", "world");
List<Integer> wordLengths = words.stream()
                                 .map(String::length)
                                 .collect(Collectors.toList());
// 输出: [5, 5]
```

对集合中的每个元素应用给定的函数。

中间操作 `Map` 能够帮助我们将 `List` 中的每一个元素做功能处理。例如下面的示例，通过 `map` 我们将每一个 `string` 转成大写：

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

2.3.3flatMap 转换

将 Stream 中的每个元素转换成另一个 Stream，然后将这些 Stream 连接起来。

将流中的每个元素映射成一个流，然后合并成一个新的流。

示例

```
List<List<Integer>> numbers = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4));
List<Integer> flattenedList = numbers.stream()
                                     .flatMap(List::stream)
                                     .collect(Collectors.toList());
// 输出: [1, 2, 3, 4]
```

2.3.4distinct(去重)

去除流中重复的元素。依赖hashcode和equals方法，所以如果是自定义类的话，就必须在自定义类里面重写hashcode和equals方法

示例

```
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 3, 4, 4, 4, 4);
List<Integer> distinctNumbers = numbers.stream()
                                       .distinct()
                                       .collect(Collectors.toList());
// 输出: [1, 2, 3, 4]
```

```

```

```
ArrayList<String> list1 = new ArrayList<>();
        list.add(”张三丰”);
        list.add(”张无忌”);
        list.add(”张翠山”);
        list.add(”王二麻子”);
        list.add(”王二麻子”);
    list.stream().distinct().forEach(s -> System.out.println(s));
 /*结果：张三丰
      张无忌
        张翠山
        王二麻子*/
```

```

### 2.3.5sorted 排序

对流中的元素进行排序，默认是自然顺序，也可以传入自定义的比较器。

`Sorted` 同样是一个中间操作，它的返参是一个 `Stream` 流。另外，我们可以传入一个 `Comparator` 用来自定义排序，如果不传，则使用默认的排序规则。

#### 示例

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5);
List<Integer> sortedNumbers = numbers.stream()
                                     .sorted()
                                     .collect(Collectors.toList());
```
```

```
// 输出: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

按照对象某个字段排序

```
        //国家按sortOrder排序
        List<AgentCountryDO> sortedCountryDOS = countryDOS.stream()
.sorted(Comparator.comparingLong(AgentCountryDO::getSortOrder))
            .collect(Collectors.toList());
```

按照倒序排序

```
        List<String> yearList =
this.lambdaQuery().select(BudgetInfoDO::getYear)
                .list()
                .stream()
                .map(BudgetInfoDO::getYear)
                .distinct()
                .sorted(Comparator.reverseOrder())
                .collect(Collectors.toList());
```

2.3.6peek方法

peek用于处理集合中元素(对象)的某个属性的值，但不改变元素(对象)的类型(区别于map操作)

示例

```
package listDemo;

import org.apache.commons.lang3.StringUtils;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class StreamPeekDemo {

    public static void main(String[] args) {

        List<Person> personList = new ArrayList<>();
        personList.add(new Person("xiaozhang", 20));
        personList.add(new Person("xiaowang", 21));

        List<Person> peekedList = personList.stream().peek(e ->
e.setCity("beijing")).collect(Collectors.toList());
        System.out.println(StringUtils.join(peekedList, "-"));

    }
}
```

```
package listDemo;

import lombok.Data;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
public class Person {

    @NonNull
    private String name;
    @NonNull
```

```
    private int age;
    private String city;

}
```

运行结果

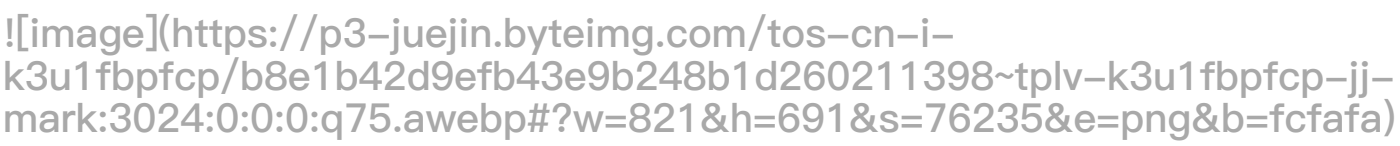
```
Person(name=xiaozhang, age=20, city=beijing)-Person(name=xiaowang, age=21, city=beijing)
```

2.3.7limit（取用前几个）

Stream limit (long maxSize): 截取指定的参数个数的数据

示例

```
ArrayList<String> list = new ArrayList<>();
        list.add(”孙悟空”);
        list.add(”唐三藏”);
        list.add(”猪八戒”);
        list.add(”沙悟净”);
//简化前代码：
 Stream<String> stream = list.stream();
stream.forEach(s -> System.out.println(s));

//简化后代码：
list.stream().limit(2).forEach(s -> System.out.println(s));
// 结果：孙悟空
//唐三藏
//注解：本方法截取就是 只保留前面的指定元素
```

```

### 2.3.8skip(跳过前几个)

跳过指定参数个数的数据 ， 就是里面这个方法里面传几，那么就会跳过前几个数据

#### 示例

```
ArrayList<String> list = new ArrayList<>();
        list.add(”孙悟空”);
        list.add(”唐三藏”);
        list.add(”猪八戒”);
        list.add(”沙悟净”);
    //Stream<T> skip (long n): 跳过指定参数个数的数据
    list.stream().skip(2).forEach(s -> System.out.println(s));
        //结果：猪八戒
        //   沙悟净
    //注解： 与截取相反 本方法 是跳过前面的指定元素
```



2.4终端操作
---
```

2.4.1collect（收集操作）

将 Stream 中的元素收集到集合或其他容器中。

示例

```
List<String> list = Arrays.asList(”a”, ”b”, ”c”);
List<String> collected = list.stream().collect(Collectors.toList());
```

注意事项：在stream流中无法直接修改集合，数组等数据源中的数据，你能修改的仅仅是流上的数据

2.4.2`Collectors`工具类

`Collectors` 是 Java 8 引入的 `java.util.stream` 包中的一个实用类，提供了多种静态方法来生成常见的收集器实例。收集器用于将流的元素处理为汇总结果，如将元素收集到一个集合、聚合元素、分组、分区等。下面是 `Collectors` 类的一些常见方法及其简介：

toList()

* 将流中的元素收集到一个 `List` 中。

```
List<String> list = stream.collect(Collectors.toList());
```

```

将流中的元素收集到一个List集合中。

```
List<Integer> collect = list.stream().filter(number -> number % 2 == 0).collect(Collectors.toList());
System.out.println(collect);
//filter 负责过滤数据的
//collect 负责收集数据的, 获取流中剩余的数据，但是他不会负责创建容器，也不负责把数据添加到容器中。
```

#### **toSet()**

* 将流中的元素收集到一个 `Set` 中，去除重复元素。

```
Set<String> set = stream.collect(Collectors.toSet());
```

```
 Set<Integer> collect1 = list.stream().filter(number -> number % 2 == 0).collect(Collectors.toSet());
 System.out.println(collect1);
//filter 负责过滤数据的
//collect 负责收集数据的, 获取流中剩余的数据，但是他不会负责创建容器，也不负责把数据添加到容器中。
```
```

toMap()

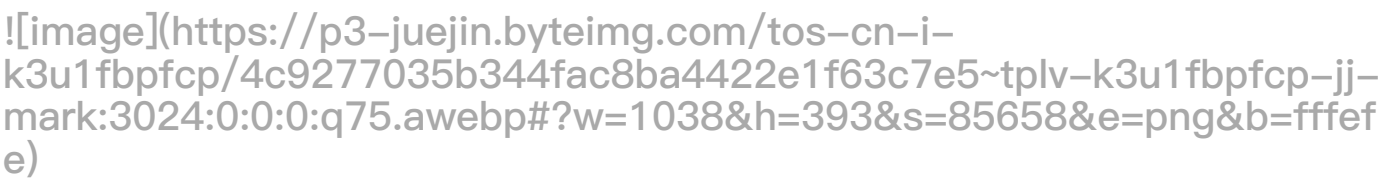
* 将流中的元素收集到一个 `Map` 中，需要指定键和值的映射函数，可以通过合并函数处理重复键。

```
Map<Integer, String> map = stream.collect(Collectors.toMap(
    String::length,  // 键映射器
    Function.identity(),  // 值映射器
    (existing, replacement) -> existing));  // 合并函数（处理重复键）
```

示例

用到了函数`Function.identity()`

```
List<GoodsSpuDO> spuDOList = this.lambdaQuery()
                .in(GoodsSpuDO::getSpuId, spuIds)
.eq(GoodsSpuDO::getLangCode,LangCodeEnum.DEFAULT.getCode())
            .eq(GoodsSpuDO::getEnableFlag,
SystemConstants.ENABLE_FLAG_ON)
                .list();
        List<GoodsSpuResourcesDO> spuResourceList =
resourcesService.lambdaQuery()
                .in(GoodsSpuResourcesDO::getSpuId, spuIds)
                .eq(GoodsSpuResourcesDO::getEnableFlag,
SystemConstants.ENABLE_FLAG_ON)
                .eq(GoodsSpuResourcesDO::getDataGroup,
GoodsSpuResourceGroupEnums.LIST_PIC.getDataGroupCode())
.in(GoodsSpuResourcesDO::getLangCode,LangCodeEnum.DEFAULT.getC
ode(),LangCodeEnum.ALL.getCode())
                .list();
        Map<Long, GoodsSpuDO> spuMap =
spuDOList.stream().collect(Collectors.toMap(GoodsSpuDO::getSpuId,
Function.identity()));
```

```



这段代码使用Java 8中的Stream API和Collectors.toMap()方法将一个包含GoodsSpuDO对象的spuDOList列表转换为一个Map\long,，其中Long是GoodsSpuDO对象的spuId属性，==GoodsSpuDO对象本身作为值==。

`Function.identity()`是一个函数引用，表示`GoodsSpuDO`对象本身作为值。

#### **joining()**

* 将流中的元素连接成一个字符串，默认使用空字符串作为分隔符。可以指定分隔符、前缀和后缀。

```
String result = stream.collect(Collectors.joining(”, ”, ”[”, ”]”));
```

将流中的元素拼接成一个字符串。

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

public class ListStreamJoiningExample {
```
```

```
    public static void main(String[] args) {
        List<String> words = Arrays.asList(”Java”, ”is”, ”a”,
”programming”, ”language”);

        // 使用joining()方法连接字符串
        String result = words.stream()
                .collect(Collectors.joining(” ”));

        // 输出结果
        System.out.println(result);
    }
}
```

在这个案例中，我们定义了一个包含多个字符串的List集合。然后，我们使用stream()方法将List转换为一个Stream对象。接着，我们调用collect(Collectors.joining(” ”))方法将Stream中的所有字符串连接起来，每个字符串之间用空格分隔。最后，我们将连接后的字符串输出到控制台。

运行上述代码，输出结果如下：

```
Java is a programming language
```

可以看到，List集合中的所有字符串都被连接起来，并以空格分隔。

mapping()

* 在收集之前应用一个额外的映射函数。

```
List<String> names = persons.stream()
    .collect(Collectors.mapping(Person::getName, Collectors.toList()));
```

mapping

`Collectors.mapping` 是一个用于收集流中元素的收集器，它可以将流中的元素进行映射操作，并将映射结果收集到指定的容器中。

`Collectors.mapping` 方法的作用是将流中的元素进行映射操作，将每个元素按照 `mapper` 函数进行转换，然后将转换后的结果传递给 `downstream` 收集器进行进一步的收集操作。

通常，`Collectors.mapping` 方法与 `Collectors.toList()`、`Collectors.toSet()` 等收集器组合使用，用于收集映射结果到列表或集合中。

例如，以下代码示例将一个字符串列表中的每个字符串转换为大写，并将结果收集到一个新的列表中：

```
List<String> strings = Arrays.asList(”apple”, ”banana”, ”orange”);
List<String> uppercaseList = strings.stream()
        .collect(Collectors.mapping(String::toUpperCase,
Collectors.toList()));
System.out.println(uppercaseList);
```

输出：

```
[APPLE, BANANA, ORANGE]
```

在这个示例中，`mapping` 方法的 `mapper` 是 `String::toUpperCase`，即将字符串转换为大写。`downstream` 是 `Collectors.toList()`，用于将映射结果收集到列表中。

注意，`Collectors.mapping` 方法是在 Java 8 中引入的，并且需要与 Java 8 或更高版本一起使用。

```
        Map<Long, List<String>> idSynonymMap =
synonymDOS.stream().collect(Collectors.groupingBy(
                SelectFunctionPointSynonymDO::getSubjectTermId,
Collectors.mapping(SelectFunctionPointSynonymDO::getSynonym, Collec
tors.toList())
        ));
```

groupingBy()

* 根据分类函数对流中的元素进行分组，生成一个 `Map`，键是分类标准，值是对应的元素列表。

```
Map<String, List<Person>> peopleByCity = people.stream()
    .collect(Collectors.groupingBy(Person::getCity));
```

Collectors.groupingBy根据一个或多个属性对集合中的项目进行分组

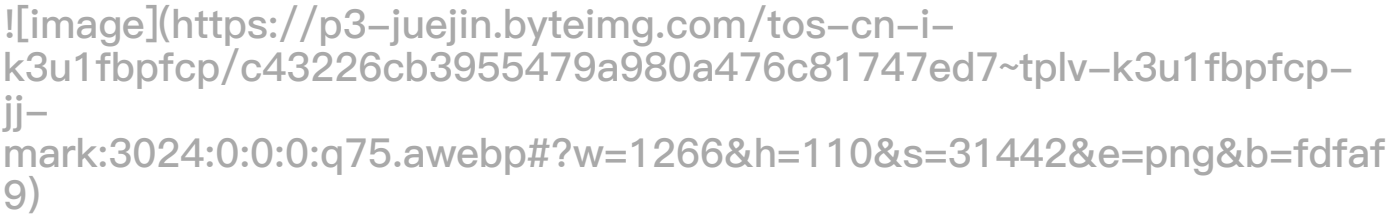
接下来这个示例，将会按年龄对所有人进行分组：

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age)); // 以年龄为 key,进行分组

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

```
@AllArgsConstructor
@NoArgsConstructor
@Data
@Builder
public class OrderDetailDTO {

    /**
     * 订单明细id
     */
    private Long id;

    /**
     * 订单id
     */
    private Long orderId;

    /**
     * 客户id
     */
    private Long customerId;

    /**
     * 同spu表中的spu_id
     */
    private Long spuId;

    /**
     * 同sku表中的sku_id
     */
```

```
    private Long skuId;
```

```

```



```

```

根据OrderDetailDTO对象列表中的spuId属性进行分组，将具有相同spuId的OrderDetailDTO对象放入同一个列表中，并将结果存储在一个Map<Long, List>对象中。

```

```

```
        Map<Long, List<String>> idSynonymMap =
synonymDOS.stream().collect(Collectors.groupingBy(
                SelectFunctionPointSynonymDO::getSubjectTermId,
Collectors.mapping(SelectFunctionPointSynonymDO::getSynonym, Collec
tors.toList())
        ));
```

```

```

reducing()

* 使用归约操作将流中的元素结合起来，类似于 `reduce()` 操作。

```

```

```
Optional<Integer> sum =
```

stream.collect(Collectors.reducing(Integer::sum));

```

#### **partitioningBy()**

* 根据谓词（条件）将流中的元素分成两组（true/false），生成一个 `Map<Boolean, List<T>>`。

```
Map<Boolean, List<Integer>> partitioned = numbers.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));
```

#### **counting()**

* 计算流中元素的数量，返回 `Long` 类型的结果。

```
long count = stream.collect(Collectors.counting());
```

summarizingInt(), summarizingDouble(), summarizingLong()

* 生成一个统计信息对象，包括最大值、最小值、平均值、总和等。

```
IntSummaryStatistics stats =
```
```

stream.collect(Collectors.summarizingInt(Integer::intValue));

```

summingLong

```
  Map<String, Long> resultData = combinedBuyingOrderDOList.stream()
                .collect(Collectors.groupingBy(e -> e.getActivityId().toString()
+ e.getSkuId().toString(),
Collectors.summingLong(CombinedBuyingOrderDO::getCommodityNum))
);

```

这段代码使用了 Java 8 中的 Stream API 和 Collectors 工具类，对一个 CombinedBuyingOrderDO 对象列表进行了分组和求和计算。

具体来说，代码中的这个 CombinedBuyingOrderDO 类包含了一些属性，如 activityId、skuId 和 commodityNum 等。现在有一个 CombinedBuyingOrderDO 对象列表 combinedBuyingOrderDOList，需要对其中的数据进行统计计算。假设列表中有多个 CombinedBuyingOrderDO 对象，它们的 activityId 和 skuId 可能相同（即表示同一种商品），而 commodityNum 表示该商品的购买数量。

代码的作用是根据 activityId 和 skuId 将 CombinedBuyingOrderDO 对象进行分组，并对同一组中的 CombinedBuyingOrderDO 对象的 commodityNum 属性进行求和操作。最终得到一个 Map 对象 resultData，其中 key 是 activityId 和 skuId 组合后的字符串，value 是同一组中 CombinedBuyingOrderDO 对象的 commodityNum 属性的和。

具体来说，代码中的 Collectors.groupingBy() 方法将 CombinedBuyingOrderDO 对象列表按照 activityId 和 skuId 进行分组，生成一个以 ”activityId + skuId” 为 key，以 CombinedBuyingOrderDO 对象的 List 为 value 的 Map 对象。
```

然后，在每个分组中使用 **Collectors.summingLong()** 方法对 CombinedBuyingOrderDO 对象的 commodityNum 属性求和。最终，得到的结果就是一个以 ”activityId + skuId” 为 key，以该组 CombinedBuyingOrderDO 对象的 commodityNum 属性值之和为 value 的 Map 对象。

minBy和maxBy

Java 8 流的新类 java.util.stream.Collectors 实现了 java.util.stream.Collector 接口，同时又提供了大量的方法对流（stream）的元素执行 map and reduce 操作，或者统计操作。

==Collectors==中的maxBy & minBy这两个函数和lambda中的max&min作用相同

```
@Test
public void maxByAndMinByExample() {
    List<String> list = Arrays.asList(”1”, ”2”, ”3”, ”4”);
    Optional<String> max = list.stream().collect(Collectors.maxBy((s, v) -> s.compareTo(v)));
    Optional<String> min = list.stream().collect(Collectors.minBy((s, v) -> s.compareTo(v)));
    System.out.println(max.get());
    System.out.println(min.get());
}
```

结果

```
```

```
4
1
```

```
@Test
public void maxAndMinExample() {
    List<String> list = Arrays.asList(”1”, ”2”, ”3”, ”4”);
    Optional<String> max = list.stream().max((s, v) -> s.compareTo(v));
    Optional<String> min = list.stream().min((s, v) -> s.compareTo(v));
    System.out.println(max.get());
    System.out.println(min.get());
}
```

结果

```
4
1
```

注意： 经过对比发现，直接使用max|min代码会更简洁、易读

2.4.3 其它

`toList()`

`.toList()` 方法用于将流（Stream）中的元素收集到一个列表中。这个方法通常用于将流转换为一个标准的 Java 集合类，如 `ArrayList` 或 `LinkedList`。

示例

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamToListExample {
    public static void main(String[] args) {
        // 创建一个字符串流
        Stream<String> stream = Stream.of(”apple”, ”banana”, ”orange”, ”kiwi”);

        // 将流中的元素收集到一个列表中
        List<String> list = stream.toList();

        // 打印列表中的元素
        System.out.println(list); // 输出: [apple, banana, orange, kiwi]
    }
}
```

.toList();与collect(Collectors.toList())的区别

`stream.toList()` 和 `collect(Collectors.toList())` 都可以将 Java 8 中的 `Stream` 转换为一个 `List` 集合，但它们有一些区别。

0. 引入方式：

* `stream.toList()` 是 `Stream` 接口的默认方法，可以直接在 `Stream` 对象上调用。
* `collect(Collectors.toList())` 是使用 `Collectors` 工具类中的静态方法，需要通过 `collect` 方法结合具体的收集器来使用。
1. 可变性：

* `stream.toList()` 返回的是一个不可变的 `List`。对返回的 `List` 进行增删操作会抛出 `UnsupportedOperationException` 异常。
* `collect(Collectors.toList())` 返回的是一个可变的 `ArrayList`。可以对返回的列表进行增删操作。
2. 需要额外的类型转换：

* `stream.toList()` 返回的是 `List` 类型，不需要进行类型转换。
* `collect(Collectors.toList())` 返回的是 `List` 接口的实现类 `ArrayList`，如果要使用 `List` 接口引用接收结果，需要进行类型转换。

综上所述，两种方式都可以将 `Stream` 转换为 `List`，但是在可变性和类型转换方面有所差异。选择哪种方式取决于具体的需求和使用场景。

toArray() ： 将 Stream 转换为数组

```
List<String> list = Arrays.asList(”a”, ”b”, ”c”);
String[] array = list.stream().toArray(String[]::new);
```

forEach（逐一处理）

对 Stream 中的每个元素执行指定操作。

示例

```
List<String> list = Arrays.asList(”a”, ”b”, ”c”);
list.stream().forEach(System.out::println);
```

```
public static class Student{
        private String name;
        private String sex;
        private String age;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

public static void main(String[] args) {
        Student student = new Student();
        student.setName(”张三”);
        student.setSex(”女”);
        student.setAge(”18”);
        Student student1 = new Student();
        student1.setName(”王美美”);
        student1.setSex(”男”);
        student1.setAge(”57”);
        List<Student> studentList = new ArrayList<>();
        studentList.add(student);
        studentList.add(student1);
        studentList.stream().forEach(stu->{
            System.out.println(stu.getName());
        });
    }
```

打印结果：张三

王美美

```

###### 集合foreach与stream foreach的区别

`forEach`和`Stream.forEach`在功能上非常相似，它们都用于遍历集合的元素并执行给定的操作。然而，它们在实现上存在一些细微的区别。

0. 使用方式：

* `forEach`是`Iterable`接口的默认方法，可以直接在集合上使用，如`list.forEach(action)`。
* `Stream.forEach`是`Stream`接口的方法，需要通过`stream()`方法将集合转换为流，然后调用`forEach(action)`，如`list.stream().forEach(action)`。
1. 可变性：

* `forEach`方法可以在遍历过程中修改原始集合的元素，因为它是直接作用于集合上的。
* `Stream.forEach`方法是一个终端操作，它对于流的每个元素执行给定的操作，但不直接修改原始集合中的元素。
2. 并行处理：

* `Stream.forEach`方法支持并行处理，可以使用`parallelStream()`方法将集合转换为并行流，从而实现并行执行给定的操作。
* `forEach`方法不直接支持并行处理，它只能按顺序逐个执行操作。

需要注意的是，尽管`Stream.forEach`方法支持并行处理，但在某些情况下，并行化操作可能会引入线程安全或同步问题。因此，在并行处理时，请确保操作是线程安全的。
```

总之，`forEach`方法是基于集合的操作，可以直接修改集合元素，而`Stream.forEach`方法是基于流的操作，不会直接修改原始集合，并且支持并行处理。根据具体的需求和场景，选择适合的方法来遍历和处理集合。

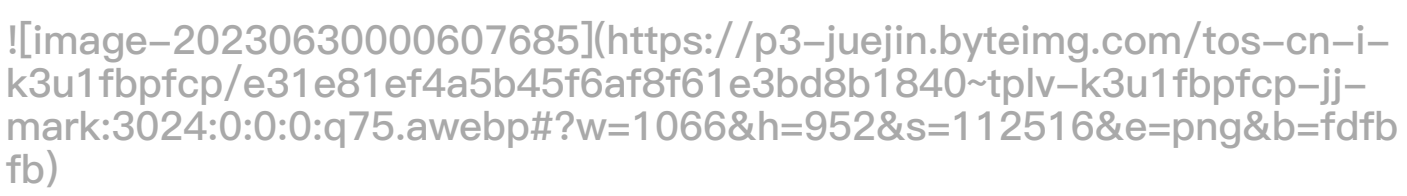
count 计数

返回 Stream 中元素的个数。

`count` 是一个终端操作，它能够统计 `stream` 流中的元素总数，返回值是 `long` 类型。

示例

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
long count = numbers.stream().count();
```



reduce

`Reduce` 中文翻译为：*减少、缩小*。通过入参的 `Function`，我们能够将 `list` 归约成一个值。它的返回类型是 `Optional` 类型。

示例

```
import java.util.stream.Stream;

public class StreamReduceExample {
    public static void main(String[] args) {
        // 创建一个整数流
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

        // 使用reduce求和
        // 这里的reduce方法将流中的元素累加起来
        // 初始值是0，accumulator是两个整数相加的操作符
        int sum = stream.reduce(0, (a, b) -> a + b);

        System.out.println("Sum: " + sum);  // 输出: Sum: 15
    }
}
```

```
Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

`findFirst`

`findFirst` 方法是 Stream API 中的一个终端操作（terminal operation），用于从 Stream 中查找并返回第一个元素。如果 Stream 为空，`findFirst` 方法将返回一个空的 `Optional` 对象。

示例

```
List<String> list = Arrays.asList("apple", "banana", "cherry");
Optional<String> firstElement = list.stream().findFirst();
```

```
List<String> myList = Arrays.asList("Java", "Kotlin", "Scala", "Groovy");
Optional<String> firstElement = myList.stream().findFirst();

// 使用 Optional 的 ifPresent 方法来处理找到的元素
firstElement.ifPresent(element -> System.out.println("The first element is: " + element));

// 如果需要获取 Optional 中的值，并处理空值情况
String first = firstElement.orElse("No elements"); // 如果没有元素，返回 "No elements"
System.out.println("The first element or default is: " + first);
```

match 匹配

Java Stream API 中的 `match` 方法是一个短路的终端操作，它用于检查 Stream 中的元素是否满足特定的条件。

0. ##### **allMatch**：

* `allMatch` 方法用于检查流中的所有元素是否都满足给定的条件。
* 它接收一个 `Predicate` 参数，返回一个 `boolean` 值。
* 如果流中的每个元素都满足条件，则返回 `true`；否则返回 `false`。
* 例如：

```
boolean allPositive = list.stream().allMatch(x -> x > 0);
```

1. ##### **anyMatch**：

* `anyMatch` 方法用于检查流中是否至少有一个元素满足给定的条件。
* 同样接收一个 `Predicate` 参数，返回一个 `boolean` 值。
* 如果流中至少有一个元素满足条件，则返回 `true`；否则返回 `false`。
* 例如：

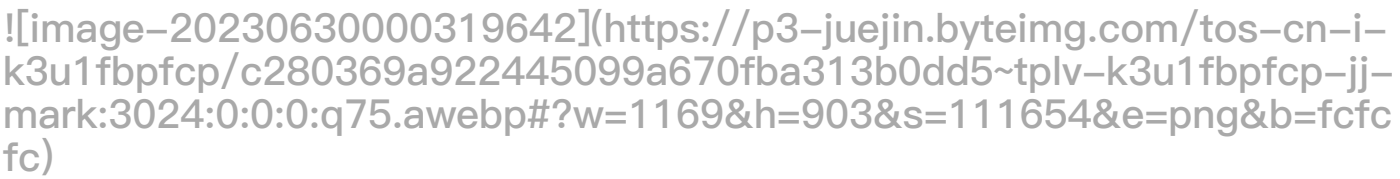
```
boolean anyNegative = list.stream().anyMatch(x -> x < 0);
```

2. ##### **noneMatch**：

* `noneMatch` 方法用于检查流中是否所有元素都不满足给定的条件。
* 也接收一个 `Predicate` 参数，返回一个 `boolean` 值。
* 如果流中没有任何元素满足条件，则返回 `true`；否则返回 `false`。
* 例如：

```
boolean noneNegative = list.stream().noneMatch(x -> x >= 0);
```

这三个方法都是短路操作，即在满足条件时会立即停止遍历流，提高了效率。



max

它用于从流中查找最大元素，具体的比较方式由传入的 `Comparator` 决定。

0. **`max(Comparator)`** ： 接受一个 `Comparator` 接口的实现作为参数，用于定义元素之间的比较逻辑。如果 Stream 不为空，返回比较后的“最大”元素；如果为空，则抛出 `NoSuchElementException`。
1. **`max()`** ： 这是 `max(Comparator)` 的简化版本，它使用元素的自然顺序来确定最大值。同样，如果 Stream 不为空，返回最大元素；如果为空，抛出 `NoSuchElementException`。

示例

使用自然顺序

```
List<Integer> numbers = Arrays.asList(3, 5, 1, 2);
Optional<Integer> maxNumber =
numbers.stream().max(Integer::compareTo);
maxNumber.ifPresent(System.out::println); // 输出最大值 5
```

使用自定义比较器

```
List<String> words = Arrays.asList("apple", "banana", "cherry");
Optional<String> longestWord =
words.stream().max(Comparator.comparingInt(String::length));
longestWord.ifPresent(System.out::println); // 输出最长的单词 "banana"
```

处理空的 Stream

```
List<Integer> emptyList = Collections.emptyList();
Optional<Integer> maxNumber =
emptyList.stream().max(Integer::compareTo);
maxNumber.ifPresentOrElse(
    System.out::println,  // 如果有最大值，打印它
    () -> System.out.println("The list is empty, no max value.") // 如果为空
，打印消息
);
```

```
maxId =
activityOrderList.stream().map(CombinedBuyingOrderDO::getId).max(Lon
g::compare).get();
```

min()： 返回 Stream 中的最小值。

示例

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5);
Optional<Integer> min = numbers.stream().min(Integer::compareTo);
```

2.5其它

2.5.1concat(合并)—静态方法

`Stream.concat` 是一个静态方法，用于将两个独立的流连接起来。

示例

```
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        Stream<Integer> stream1 = Stream.of(1, 2, 3);
        Stream<Integer> stream2 = Stream.of(4, 5, 6);

        Stream<Integer> concatenatedStream = Stream.concat(stream1,
```

```
stream2);
        concatenatedStream.forEach(System.out::println);
        // 输出: 1, 2, 3, 4, 5, 6
    }
}

```

static Stream concat (Stream a,Stream b): 合并a 和 b两个流为一个流

```
ArrayList<String> list = new ArrayList<>();
list.add(”张三丰”);
list.add(”张无忌”);
list.add(”张翠山”);
list.add(”王二麻子”);

ArrayList<String> list2 = new ArrayList<>();
list2.add(”张三丰夫人”);
list2.add(”张无忌夫人”);
list2.add(”张翠山夫人”);
list2.add(”王二麻子夫人”);
list2.add(”谢广坤夫人”);
list2.add(”张良夫人”);
//简化前
Stream<String> stream1 = list.stream();
Stream<String> stream2 = list2.stream();

Stream<String> stream3 = Stream.concat(stream1, stream2);
stream3.forEach(s -> System.out.println(s));

//简化后
Stream.concat(list.stream(),list2.stream()).forEach(s ->
System.out.println(s));

```