巨将魔法书--Stream

1. Stream API
    1.1. filter(Predicate predicate)：
    1.2. map(Function mapper)：
    1.3. flatMap(Function mapper)：
    1.4. peek(Consumer action)：
    1.5. distinct()：
    1.6. sorted()：
    1.7. sorted(Comparator comparator)：
    1.8. limit(long maxSize)：
    1.9. skip(long n)：
    1.10. reduce(BinaryOperator accumulator)：
    1.11. collect(Collector collector)：
    1.12. allMatch(Predicate predicate)：
    1.13. anyMatch(Predicate predicate)：
    1.14. noneMatch(Predicate predicate)：
    1.15. findFirst()：
    1.16. findAny()：
    1.17. count()：
    1.18. forEach(Consumer action)：
2. Collector API
    2.1. toList()
    2.2. toSet()
    2.3. toCollection(Supplier<C> collectionFactory)
    2.4. counting()
    2.5. summingInt(ToIntFunction<? super T> mapper)
    2.6. averagingInt(ToIntFunction<? super T> mapper)
    2.7. summarizingInt(ToIntFunction<? super T> mapper)
    2.8. joining()

2.9. maxBy(Comparator<? super T> comparator)
2.10. minBy(Comparator<? super T> comparator)
2.11. groupingBy(Function<? super T,? extends K> classifier)
2.12. partitioningBy(Predicate<? super T> predicate)
2.13. mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)
2.14. reducing(BinaryOperator<T> op)
2.15. collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)
2.16. summingLong(ToLongFunction<? super T> mapper)
2.17. averagingLong(ToLongFunction<? super T> mapper):
2.18. summarizingLong(ToLongFunction<? super T> mapper)
2.19. toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMap...
2.20. toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U...
2.21. groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)
2.22. groupingByConcurrent(Function<? super T,? extends K> classifier)
2.23. groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> do...
2.24. reducing(T identity, BinaryOperator<T> op)
2.25. reducing(T identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)
2.26. groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)
2.27. groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? su...
2.28. groupingByConcurrent(Function<? super T,? extends K> classifier)
2.29. groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> do...
2.30. groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Col...
3. 踩坑事项
3.1. 注意Stream的消费性
3.2. 避免在forEach中修改源集合：
3.3. 小心使用并行Stream：
3.4. 避免在Stream中改变状态：
3.5. 注意Optional的使用：
3.6. 避免长时间运行的操作：
3.7. 注意null值：
3.8. 小心使用reduce操作：
3.9. 不要忽视peek操作：

3.10. 注意flatMap操作：
3.11. 避免使用Stream.iterate生成大量数据：
3.12. 注意Stream的惰性求值特性：
3.13. 避免在并行流中使用有状态的函数：
3.14. 小心处理无限Stream：
3.15. 注意处理异常：
3.16. 避免在Stream中使用break或return：
3.17. 避免混淆map和flatMap：
3.18. 小心处理boxed类型：
3.19. 注意Stream不是集合：
3.20. 使用适当的数据结构：
3.21. 小心处理空Stream：
3.22. 注意collect操作的性能：
3.23. 避免在Stream操作中改变输入参数：
3.24. 避免在并行Stream中使用线程不安全的数据结构：
3.25. 避免在Stream中更改输入源：
3.26. 使用适当的Collector：
3.27. 避免在并行Stream中使用有序Collector：
3.28. 避免在Stream中使用阻塞操作：
3.29. 注意parallel和sequential的影响范围：
3.30. 避免在Stream中使用有副作用的函数：
3.31. 避免在Stream中使用Random：
3.32. 小心处理异常：
3.33. 避免使用不适合并行处理的操作：
3.34. 避免使用不适合的数据类型：
3.35. 注意并行Stream的线程使用：
3.36. 小心处理异常：
3.37. 避免在Stream中使用全局变量：
3.38. 注意toArray方法的使用：
4. 踩坑的测试用例
4.1. 测试Stream的消费性：
4.2. 测试在forEach中修改源集合：

4.3. 测试在并行Stream中使用线程不安全的数据结构：

4.4. 测试在Stream中使用null值：

4.5. 测试Optional的使用：

4.6. 测试在map中更改输入参数：

4.7. 测试并行Stream的线程使用：

4.8. 测试处理异常：

4.9. 测试在Stream中使用全局变量：

4.10. 测试toArray方法的使用：

4.11. 测试在Stream中使用有副作用的函数：

4.12. 测试在Stream中使用Random：

4.13. 测试在Stream中处理空Stream：

4.14. 测试collect操作的性能：

4.15. 测试在Stream中使用阻塞操作：

4.16. 测试reduce操作的恒等值：

4.17. 测试peek操作的调试用途：

4.18. 测试flatMap操作的内存占用：

4.19. 测试Stream.iterate生成大量数据：

4.20. 测试处理无限Stream：

4.21. 测试reduce操作的使用：

4.22. 测试peek操作的使用：

4.23. 测试flatMap操作的使用：

4.24. 测试Stream.iterate的使用：

4.25. 测试Stream的惰性求值特性：

4.26. 测试boxed类型：

4.27. 测试Stream不是集合：

4.28. 测试使用适当的数据结构：

4.29. 测试在并行Stream中使用有状态的函数：

4.30. 测试在Stream中使用break或return：

4.31. 测试collect操作的性能：

4.32. 测试在Stream操作中改变输入参数：

4.33. 测试在并行Stream中使用线程不安全的数据结构：

4.34. 测试在Stream中使用null值：

4.35. 测试Optional的使用：

> Java中的Stream API是从Java 8开始引入的，用于处理集合对象。一个Stream就像一个迭代器（Iterator），但是Stream API更为强大，因为它提供了并行处理的能力。
>
> Stream API可以对集合数据进行非常复杂的查找、过滤和映射数据等操作。Stream操作分为中间操作或者最终操作两种，最终操作返回一特定类型的计算结果，而中间操作返回Stream本身，这样你就可以将多个操作依次串起来。

1. Stream API

> 在Java中，Stream API 提供了大量的操作方法，以便在集合上执行复杂的数据处理任务。以下是一些常用的 Stream 操作：

1.1. filter(Predicate predicate)：

返回一个新的Stream，其中包含原Stream中所有满足predicate条件的元素。例如，`stream.filter(e -> e > 10)`会返回一个新的Stream，其中包含原Stream中所有大于10的元素。

1.2. map(Function mapper)：

返回一个新的Stream，其中包含将原Stream中的每个元素通过mapper函数处理后的结果。例如，`stream.map(e -> e * e)`会返回一个新的Stream，其中包含原Stream中每个元素的平方。

1.3. flatMap(Function mapper)：

类似于map，但是mapper函数的返回值必须是一个Stream。flatMap会将原Stream中所有元素取出放入到一个新的Stream中。例如，`stream.flatMap(e -> Stream.of(e, e * 10))`会返回一个新的Stream，其中包含原Stream中每个元素以及对应元素的10倍。

1.4. peek(Consumer action)：

返回一个新的Stream，其中包含原Stream中所有元素，但是在返回之前，会先将每个元素传递给action进行处理。例如，`stream.peek(e -> System.out.println(e))`会打印出原Stream中的每个元素。

1.5. distinct()：

返回一个新的Stream，其中包含原Stream中所有不重复的元素。

1.6. sorted()：

返回一个新的Stream，其中包含原Stream中所有元素，但是元素按照自然顺序进行排序。

1.7. sorted(Comparator comparator)：

返回一个新的Stream，其中包含原Stream中所有元素，但是元素按照提供的Comparator进行排序。

1.8. limit(long maxSize)：

返回一个新的Stream，其中包含原Stream中的前maxSize个元素。

1.9. skip(long n)：

返回一个新的Stream，其中包含原Stream中除前n个元素外的所有元素。

1.10. reduce(BinaryOperator accumulator)：

返回一个Optional，其中包含用accumulator迭代地将原Stream中的所有元素组合起来得到的结果。例如，`stream.reduce((a, b) -> a + b)`会返回一个Optional，其中包含原Stream中所有元素的和。

1.11. collect(Collector collector)：

返回一个新的集合，其中包含将原Stream中的所有元素收集到collector指定的结果容器中。例如，`stream.collect(Collectors.toList())`会返回一个List，其中包含原Stream中的所有元素。

1.12. allMatch(Predicate predicate)：

返回一个boolean值，表示原Stream中的所有元素是否都满足predicate指定的条件。

1.13. anyMatch(Predicate predicate)：

返回一个boolean值，表示原Stream中是否存在至少一个元素满足predicate指定的条件。

1.14. noneMatch(Predicate predicate)：

返回一个boolean值，表示原Stream中是否不存在任何一个元素满足predicate指定的条件。

1.15. findFirst()：

返回一个Optional，其中包含原Stream中的第一个元素。

1.16. findAny()：

返回一个Optional，其中包含原Stream中的任意一个元素。

1.17. count()：

返回一个long值，表示原Stream中的元素个数。

1.18. forEach(Consumer action)：

对原Stream中的每个元素执行action指定的操作。例如，`stream.forEach(e -> System.out.println(e))`会打印出原Stream中的每个元素。

---

2. Collector API

> 在Java的Stream API中，Collector是一个接口，它定义了将元素累积到某种结果容器中的多种方法。它提供了一种通用的、可并行的、可组合的数据处理能力。

Collector接口中定义了一些方法，例如supplier()用于生成一个新的结果容器，accumulator()用于将输入元素添加到结果容器中，combiner()用于将两个结果容器合并为一个，finisher()用于完成收集操作并生成最终结果。

Java提供了Collectors类，它实现了Collector接口并提供了许多用于从Stream生成集合、值、字符串等常见类型的静态工厂方法。例如，Collectors.toList()可以将Stream的元素收集到一个List中，Collectors.joining()可以将Stream的元素连接为一个字符串。

2.1. toList()

返回一个Collector，它将输入元素累积到一个新的List中。

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

2.2. toSet()

返回一个Collector，它将输入元素累积到一个新的Set中。

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

```

2.3. toCollection(Supplier<C> collectionFactory)

返回一个Collector，它将输入元素累积到由指定的Supplier返回的Collection中。

```java
Collection<String> collection = stream.collect(Collectors.toCollection(ArrayList::new));

```

2.4. counting()

返回一个Collector，它计算输入元素的个数。

```java
Long count = stream.collect(Collectors.counting());

```

2.5. summingInt(ToIntFunction<? super T> mapper)

返回一个Collector，它将输入元素映射为提供的mapper然后返回其总和。

```java
Integer sum = stream.collect(Collectors.summingInt(Integer::valueOf));

```

2.6. averagingInt(ToIntFunction<? super T> mapper)

返回一个Collector，它将输入元素映射为提供的mapper然后返回其平均值。

```java
Double average = stream.collect(Collectors.averagingInt(Integer::valueOf));

```

2.7. summarizingInt(ToIntFunction<? super T> mapper)

返回一个Collector，它将输入元素映射为提供的mapper然后返回其总和、平均值、最大值和最小值。

```java
Double average = stream.collect(Collectors.averagingInt(Integer::valueOf));

```

2.8. joining()

返回一个Collector，它连接对输入元素的字符串表示形式，由逗号（,）分隔。

```java
String joined = stream.collect(Collectors.joining(", "));

```

2.9. maxBy(Comparator<? super T> comparator)

返回一个Collector，它根据提供的比较器计算输入元素的最大值。

```java
Optional<String> max = stream.collect(Collectors.maxBy(Comparator.naturalOrder()));

```

2.10. minBy(Comparator<? super T> comparator)

返回一个Collector，它根据提供的比较器计算输入元素的最小值。

```java
Optional<String> min = stream.collect(Collectors.minBy(Comparator.naturalOrder()));

```

2.11. groupingBy(Function<? super T,? extends K> classifier)

返回一个Collector，它根据分类函数将输入元素分组到Map中。

```java
Map<String, List<String>> grouped = stream.collect(Collectors.groupingBy(Function.identity()));
```

2.12. partitioningBy(Predicate<? super T> predicate)

返回一个Collector，它根据给定的Predicate将输入元素分区到一个Map中。例如：

```
Map<Boolean, List<String>> partitioned = stream.collect(Collectors.partitioningBy(s -> s.length() > 5));
```

2.13. mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)

返回一个Collector，它将输入元素映射到一个新的类型，并使用给定的Collector进行进一步的归约。例如：

```
List<Integer> mapped = stream.collect(Collectors.mapping(String::length, Collectors.toList()));
```

2.14. reducing(BinaryOperator<T> op)

返回一个Collector，它使用给定的二元运算符对输入元素进行归约。例如：

```
Optional<String> reduced = stream.collect(Collectors.reducing((s1, s2) -> s1 + "#" + s2));
```

2.15. collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)

返回一个Collector，它首先使用给定的Collector对输入元素进行归约，然后使用给定的函数对结果进行转换。例如：

```
List<String> collectedThenTransformed = stream.collect(Collectors.collectingAndThen(Collectors.toList(), list -> list.subList(0, 5)));
```

2.16. summingLong(ToLongFunction<? super T> mapper)

返回一个Collector，它将输入元素映射为提供的mapper然后返回其总和。例如：

```
Plain Text
1  Long sum = stream.collect(Collectors.summingLong(Long::valueOf));
```

2.17. averagingLong(ToLongFunction<? super T> mapper):

返回一个Collector，它将输入元素映射为提供的mapper然后返回其平均值。例如：

```
Plain Text
1  Double average = stream.collect(Collectors.averagingLong(Long::valueOf));
```

2.18. summarizingLong(ToLongFunction<? super T> mapper)

返回一个Collector，它将输入元素映射为提供的mapper然后返回其总和、平均值、最大值和最小值。例如：

```
Plain Text
1  LongSummaryStatistics statistics = stream.collect(Collectors.summarizingLon
   g(Long::valueOf));
```

2.19. toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)

返回一个Collector，它将输入元素累积到一个Map中，其中的键和值是由给定的函数生成的。例如：

```
Plain Text
1  Map<String, Integer> map = stream.collect(Collectors.toMap(Function.identit
   y(), String::length));
```

2.20. toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)

返回一个Collector，它将输入元素累积到一个ConcurrentMap中，其中的键和值是由给定的函数生成的。例如：

```
Plain Text
1  ConcurrentMap<String, Integer> concurrentMap = stream.collect(Collectors.to
   ConcurrentMap(Function.identity(), String::length));
```

2.21. groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

返回一个Collector，它根据分类函数将输入元素分组到Map中，并将分类后的结果进行进一步的归约。例如：

```
Plain Text
1  Map<String, Long> groupedCount = stream.collect(Collectors.groupingBy(Funct
   ion.identity(), Collectors.counting()));
```

2.22. groupingByConcurrent(Function<? super T,? extends K> classifier)

返回一个Collector，它根据分类函数将输入元素分组到ConcurrentMap中。例如：

```
Plain Text
1  ConcurrentMap<String, List<String>> groupedConcurrent = stream.collect(Coll
   ectors.groupingByConcurrent(Function.identity()));
```

2.23. groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

返回一个Collector，它根据分类函数将输入元素分组到ConcurrentMap中，并将分类后的结果进行进一步的归约。例如：

```
Plain Text
1  ConcurrentMap<String, Long> groupedCountConcurrent = stream.collect(Collect
   ors.groupingByConcurrent(Function.identity(), Collectors.counting()));
```

2.24. reducing(T identity, BinaryOperator<T> op)

返回一个Collector，它使用给定的二元运算符对输入元素进行归约，结果是Optional，其中包含归约的结果。例如：

```
Plain Text
1  String reduced = stream.collect(Collectors.reducing("", (s1, s2) -> s1 + "#" + s2));
```

2.25. reducing(T identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)

返回一个Collector，它首先使用映射函数转换输入元素，然后使用给定的二元运算符进行归约。例如：

```
Java
1  Integer reduced = stream.collect(Collectors.reducing(0, String::length, Integer::sum));
```

2.26. groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

返回一个Collector，它根据分类函数将输入元素分组到Map中，并对每个分组的元素进行进一步的归约。例如：

```
Plain Text
1  Map<String, Long> grouped = stream.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

2.27. groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)

返回一个Collector，它根据分类函数将输入元素分组到由给定的Supplier产生的Map中，并对每个分组的元素进行进一步的归约。例如：

```
Plain Text
1  Map<String, Long> grouped = stream.collect(Collectors.groupingBy(Function.identity(), TreeMap::new, Collectors.counting()));
```

2.28. groupingByConcurrent(Function<? super T,? extends K> classifier)

返回一个Collector，它根据分类函数将输入元素分组到ConcurrentMap中。例如：

```
Plain Text
1  ConcurrentMap<String, List<String>> grouped = stream.collect(Collectors.gro
   upingByConcurrent(Function.identity()));
```

2.29. groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

返回一个Collector，它根据分类函数将输入元素分组到ConcurrentMap中，并对每个分组的元素进行进一步的归约。例如：

```
Plain Text
1  ConcurrentMap<String, Long> grouped = stream.collect(Collectors.groupingByC
   oncurrent(Function.identity(), Collectors.counting()));
```

2.30. groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)

返回一个Collector，它根据分类函数将输入元素分组到由给定的Supplier产生的ConcurrentMap中，并对每个分组的元素进行进一步的归约。例如：

```
Plain Text
1  ConcurrentMap<String, Long> grouped = stream.collect(Collectors.groupingByC
   oncurrent(Function.identity(), ConcurrentSkipListMap::new, Collectors.count
   ing()));
```

3. 踩坑事项

> 当你使用Java的Stream API时，有一些常见的陷阱和禁忌需要避免：

3.1. 注意Stream的消费性

Stream只能被消费一次。一旦你调用了任何终止操作，如collect、count、forEach等，Stream就被“消费”了，不能再被使用。如果你试图再次使用这个Stream，会抛出java.lang.IllegalStateExceptionstream has already been operated upon or closed异常。

3.2. 避免在forEach中修改源集合：

在使用forEach进行元素操作时，应避免修改源集合。因为这可能会导致ConcurrentModificationException异常或者得到不可预期的结果。

3.3. 小心使用并行Stream：

虽然并行Stream可以提高处理大数据集的效率，但它也会增加代码的复杂性，因为你需要处理线程安全问题。并且，并行Stream并不总是比顺序Stream更快，特别是在数据量较小的情况下。

3.4. 避免在Stream中改变状态：

你应该避免在Stream的操作中改变状态，特别是在并行Stream中。这是因为Stream的操作可能会在多个线程中并行执行，改变状态可能会导致数据竞争和不一致的结果。

3.5. 注意Optional的使用：

findFirst、findAny等方法返回的是Optional对象，而不是集合中的元素。你需要调用Optional的get()或orElse()方法来获取实际的值。直接调用Optional.get()可能会抛出NoSuchElementException，所以最好在调用get()之前使用isPresent()检查是否有值。

3.6. 避免长时间运行的操作：

Stream操作应该是轻量级的，快速完成。在Stream操作中进行网络调用、I/O操作或者CPU密集型计算可能会导致Stream处理非常慢，特别是在并行Stream中，这可能会导致其他线程被阻塞，降低整体性能。

3.7. 注意null值：

许多Stream操作不接受null值，如果Stream中的元素或者你提供的函数返回null，可能会抛出NullPointerException。你需要确保你的Stream中不包含null值，或者在进行操作前先进行null检查。

3.8. 小心使用reduce操作：

reduce操作需要提供一个恒等值（identity），这个值在流为空的时候会被返回，同时它还应该是组合函数的单位元素。如果你提供的恒等值或者组合函数不满足这些要求，可能会得到错误的结果。

3.9. 不要忽视peek操作：

peek操作主要用于调试，它可以让你查看Stream中的值而不改变它们。但是如果你忘记了peek操作，可能会看到一些奇怪的输出。

3.10. 注意flatMap操作：

flatMap操作可以将多个Stream连接成一个Stream，但是如果每个元素生成的Stream元素非常多，可能会导致内存占用过高。在使用flatMap时，你需要确保生成的Stream大小是可控的。

3.11. 避免使用Stream.iterate生成大量数据：

Stream.iterate可以生成一个无限的Stream，如果你尝试在这个Stream上进行终止操作，如count、collect等，可能会导致程序永不停止，或者内存溢出。

3.12. 注意Stream的惰性求值特性：

Stream的许多操作是惰性的，只有在真正需要结果的时候才会执行。这意味着如果你没有对Stream进行终止操作，那么中间操作实际上是不会执行的。例如，下面的代码实际上并不会打印任何东西：

```java
Stream.of("one", "two", "three", "four")
.filter(e -> {
    System.out.println("Filtering " + e);
    return true;
});
```

你需要添加一个终止操作（如forEach或collect）来触发计算。

3.13. 避免在并行流中使用有状态的函数：

如果你在并行Stream的map或filter等操作中使用有状态的函数，可能会导致结果不正确。因为在并行处理的时候，函数可能会被多个线程同时调用，如果函数内部有状态（比如修改了一个全局变量），可能会导致数据竞争和不一致的结果。

3.14. 小心处理无限Stream：

Stream.iterate和Stream.generate可以创建无限的Stream，这些Stream没有固定的大小或结束。如果你对这样的Stream进行count、collect或reduce等需要处理整个Stream的操作，程序可能会进入无限循环。你应该在无限Stream上使用limit等操作来限制其大小。

3.15. 注意处理异常：

在Stream的map或filter等操作中，你通常不能直接抛出受检异常（checked exception），因为这些函数式接口的方法没有声明任何异常。你需要使用try/catch来处理异常，或者将受检异常转换为非受检异常。

3.16. 避免在Stream中使用break或return：

在Stream的forEach或其他方法的Lambda表达式中，你不能使用break或return来提前结束处理或跳过某些元素。这是因为Lambda表达式实际上是一个函数，break或return会导致函数立即返回，而不是结束Stream处理。如果你需要提前结束处理或跳过某些元素，你应该使用filter或takeWhile、dropWhile（Java 9中新增的方法）等操作。

3.17. 避免混淆map和flatMap：

map和flatMap都可以用来转换Stream中的元素，但是它们的作用是不同的。map方法是将一个元素转换为另一个元素，而flatMap方法是将一个元素转换为一个Stream，然后将多个Stream连接成一个Stream。当你需要将多个集合或数组合并成一个时，应该使用flatMap。

3.18. 小心处理boxed类型：

IntStream、LongStream和DoubleStream等原始类型Stream比通常的Stream<Integer>、Stream<Long>和Stream<Double>更为高效，因为它们避免了自动装箱和拆箱的开销。但是，如果你需要将原始类型Stream转换为对象类型Stream，或者需要使用对象类型特有的方法，你需要使用boxed方法来进行转换。

3.19. 注意Stream不是集合：

虽然Stream可以用来处理集合，但是它本身并不是一种集合，它不是数据结构并且不能被重复使用或者并发修改。你应该避免将Stream作为方法的返回类型，或者在类中持有一个Stream字段。

3.20. 使用适当的数据结构：

并非所有情况都适合使用Stream。在处理大量数据时，使用适当的数据结构（如数组或集合）可能会比使用Stream更为高效。你应该根据实际情况选择最适合的工具。

3.21. 小心处理空Stream：

当你处理一个可能为空的Stream时，你需要注意findFirst、findAny、reduce等操作返回的是Optional，而不是具体的值。你需要调用Optional的get()或orElse()方法来获取值。直接调用Optional.get()可能会抛出NoSuchElementException，所以最好在调用get()之前使用isPresent()检查是否有值。

3.22. 注意collect操作的性能：

collect操作可以将Stream的元素收集到一个集合中，但是这需要在内存中存储所有的元素。如果Stream的元素非常多，可能会导致内存溢出。在处理大数据集时，你应该使用reduce或forEach等操作，而不是collect。

3.23. 避免在Stream操作中改变输入参数：

在Stream的map、filter等操作中，你应该避免改变输入参数。这些操作应该是纯函数（pure function），即对于相同的输入，总是产生相同的输出，而且不产生任何副作用。

3.24. 避免在并行Stream中使用线程不安全的数据结构：

如果你在并行Stream的map或collect等操作中使用线程不安全的数据结构，可能会导致数据竞争和不一致的结果。你应该使用Collections.synchronizedList、Collections.synchronizedSet、Collections.synchronizedMap等线程安全的数据结构，或者使用java.util.concurrent包中的数据结构。

3.25. 避免在Stream中更改输入源：

在使用Stream时，你应该避免更改输入源（如集合或数组）。在Stream操作中更改输入源可能会导致不可预测的结果或并发修改异常。例如，在filter或map等操作中添加或删除集合元素是不安全的。

3.26. 使用适当的Collector：

collect方法可以使用Collector将Stream元素收集到集合或其他数据结构。你应该使用适当的Collector来满足你的需求。例如，如果你需要一个Set，你应该使用Collectors.toSet()而不是Collectors.toList()。

3.27. 避免在并行Stream中使用有序Collector：

有些Collector，如Collectors.joining()和Collectors.toList()，会保留元素的顺序。在并行Stream中使用这些Collector可能会降低性能，因为它需要在所有线程之间同步元素的顺序。如果你不需要保留元素的顺序，你应该使用无序的Collector，如Collectors.toSet()。

3.28. 避免在Stream中使用阻塞操作：

在Stream操作中执行阻塞操作，如网络I/O或线程睡眠，可能会导致整个Stream处理阻塞。你应该在Stream外部执行这些操作，或者使用异步API。

3.29. 注意parallel和sequential的影响范围：

parallel和sequential方法可以将Stream切换为并行或顺序模式。它们的影响范围是从调用位置到Stream的末端（或下一个parallel或sequential调用）。你应该避免在同一个Stream中多次切换并行和顺序模式，因为这可能会导致混乱和性能下降。

3.30. 避免在Stream中使用有副作用的函数：

在Stream的map、filter等操作中，你应该避免使用有副作用的函数。这些操作应该是纯函数（pure function），即对于相同的输入，总是产生相同的输出，而且不产生任何副作用。例如，你不应该在map操作中修改一个全局变量。

3.31. 避免在Stream中使用Random：

Random对象不是线程安全的，如果你在并行Stream中使用同一个Random对象生成随机数，可能会导致数据竞争和不一致的结果。你应该使用ThreadLocalRandom或者为每个线程创建一个新的Random对象。

3.32. 小心处理异常：

在Stream的map或filter等操作中，你通常不能直接抛出受检异常（checked exception），因为这些函数式接口的方法没有声明任何异常。你需要使用try/catch来处理异常，或者将受检异常转换为非受检异常。

3.33. 避免使用不适合并行处理的操作：

并非所有的Stream操作都适合并行处理。例如，limit、findFirst、findAny等依赖于元素的顺序的操作，在并行Stream中可能会导致性能下降，因为它们需要等待所有元素处理完毕。在使用并行Stream时，你应该主要使用map、filter、forEach等不依赖于元素顺序的操作。

3.34. 避免使用不适合的数据类型：

虽然Stream API可以处理各种类型的数据，但并非所有的数据类型都适合使用Stream。例如，处理原始数据类型（如int、long等）时，使用特殊的IntStream、LongStream等会比标准的Stream<Integer>、Stream<Long>等更高效，因为它们避免了自动装箱和拆箱的开销。

3.35. 注意并行Stream的线程使用：

并行Stream使用公共的ForkJoinPool，它默认的线程数量等于你的处理器数量。如果你的并行Stream操作是CPU密集型的，增加线程数量可能无法提高性能，反而会因为线程切换带来额外的开销。另外，如果你在并行Stream操作中执行阻塞操作，可能会导致其他并行Stream饿死（即无法获得执行的机会）。

3.36. 小心处理异常：

Stream的lambda表达式不能抛出受检异常，你需要在lambda表达式内部捕获异常。如果你的操作抛出受检异常，你可能需要将它封装为运行时异常，或者使用try/catch块处理它。

3.37. 避免在Stream中使用全局变量：

在Stream的操作中，你应该避免使用全局变量，因为这可能导致线程安全问题。如果你需要在多个Stream操作之间共享状态，你应该使用reduce或collect方法，而不是全局变量。

3.38. 注意toArray方法的使用：

toArray方法可以将Stream转换为数组，但你需要提供一个生成器函数来创建正确类型的数组。如果你直接使用toArray()，它将返回一个Object[]，你可能需要进行类型转换。

4. 踩坑的测试用例

> 以下是一些基于上述禁忌事项的单元测试用例：

4.1. 测试Stream的消费性：

验证一旦Stream被消费，再次使用会抛出异常。

```java
@Test(expected = IllegalStateException.class)
public void testStreamConsumption() {
    Stream<String> stream = Stream.of("one", "two", "three");
    stream.count();
    stream.collect(Collectors.toList());  // This should throw an exception
}
```

4.2. 测试在forEach中修改源集合：

验证在forEach中修改源集合会抛出异常。

```java
@Test(expected = ConcurrentModificationException.class)
public void testModifyingSourceInForEach() {
    List<String> list = new ArrayList<>(Arrays.asList("one", "two", "three"));
    list.stream().forEach(e -> list.remove(e));  // This should throw an exception
}
```

4.3. 测试在并行Stream中使用线程不安全的数据结构：

验证在并行Stream中使用线程不安全的数据结构可能会导致数据竞争和不一致的结果。

```java
@Test
public void testUnsafeParallelStream() {
    List<Integer> list = new ArrayList<>();
    IntStream.range(0, 10000).parallel().forEach(list::add);
    assertNotEquals(10000, list.size());  // The size may not be 10000 due to data race
}
```

4.4. 测试在Stream中使用null值：

验证在Stream中使用null值可能会抛出NullPointerException。

```java
@Test(expected = NullPointerException.class)
public void testNullInStream() {
    Stream.of("one", null, "three").map(String::length).collect(Collectors.toList());  // This should throw an exception
}
```

4.5. 测试Optional的使用：

验证findFirst、findAny等返回的是Optional，而不是实际的值。

```java
@Test
public void testOptionalInStream() {
    Stream<String> stream = Stream.of("one", "two", "three");
    Optional<String> optional = stream.findFirst();
    assertTrue(optional.isPresent());
    assertEquals("one", optional.get());
}
```

4.6. 测试在map中更改输入参数：

验证在map中更改输入参数可能会导致错误的结果。

```java
@Test
public void testModifyingInputInMap() {
    List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3));
    list.stream().map(e -> e = e + 1).collect(Collectors.toList());
    assertEquals(Arrays.asList(1, 2, 3), list);  // The original list should not be changed
}
```

4.7. 测试并行Stream的线程使用：

验证并行Stream使用公共的ForkJoinPool，并且默认的线程数量等于处理器数量。

```java
@Test
public void testParallelStreamThreadUsage() {
    long count = Stream.of(1, 2, 3, 4, 5).parallel()
    .peek(e -> assertTrue(Thread.currentThread().getName().startsWith("ForkJoinPool")))
    .count();
    assertEquals(5, count);
}
```

4.8. 测试处理异常：

验证在Stream的map或filter等操作中，不能直接抛出受检异常。

```java
@Test(expected = RuntimeException.class)
public void testExceptionHandlingInStream() {
    Stream.of("1", "2", "invalid").map(Integer::parseInt).collect(Collectors.toList());  // This should throw an exception
}
```

4.9. 测试在Stream中使用全局变量：

验证在Stream中使用全局变量可能会导致线程安全问题。

```java
@Test
public void testGlobalVariableInStream() {
    List<Integer> list = new ArrayList<>();
    IntStream.range(0, 10000).parallel().forEach(list::add);
    assertNotEquals(10000, list.size());  // The size may not be 10000 due to data race
}
```

4.10. 测试toArray方法的使用：

验证toArray方法返回的是Object[]，需要进行类型转换。

```java
@Test
public void testToArrayInStream() {
    Object[] array = Stream.of("one", "two", "three").toArray();
    assertTrue(array instanceof Object[]);
    assertEquals("one", array[0]);
}
```

4.11. 测试在Stream中使用有副作用的函数：

验证在Stream中使用有副作用的函数可能会导致错误的结果。

```java
@Test
public void testSideEffectInStream() {
    List<String> list = new ArrayList<>();
    Stream.of("one", "two", "three").filter(e -> {
        list.add(e);
        return true;
    }).collect(Collectors.toList());
    assertTrue(list.isEmpty());  // The list should still be empty because the filter operation is lazy
}
```

4.12. 测试在Stream中使用Random：

验证在并行Stream中使用同一个Random对象生成随机数可能会导致数据竞争和不一致的结果。

```java
@Test
public void testRandomInParallelStream() {
    Random random = new Random();
    Set<Integer> set = IntStream.range(0, 10000).parallel()
    .map(i -> random.nextInt()).boxed().collect(Collectors.toSet());
    assertTrue(set.size() < 10000);  // The size may not be 10000 due to data race
}
```

4.13. 测试在Stream中处理空Stream：

验证在处理可能为空的Stream时，findFirst、findAny等返回的是Optional，而不是实际的值。

```java
@Test
public void testEmptyStream() {
    Stream<String> stream = Stream.empty();
    Optional<String> optional = stream.findFirst();
    assertFalse(optional.isPresent());
}
```

4.14. 测试collect操作的性能：

验证collect操作可能会导致内存溢出。

```java
@Test(timeout = 1000)
public void testCollectPerformance() {
    List<Integer> list = IntStream.range(0, 10000000).boxed().collect(Collectors.toList());
    assertEquals(10000000, list.size());
}
```

4.15. 测试在Stream中使用阻塞操作：

验证在Stream操作中执行阻塞操作可能会导致整个Stream处理阻塞。

```java
@Test(timeout = 1000)
public void testBlockingOperationInStream() {
    Stream.of("one", "two", "three").map(e -> {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
        return e.toUpperCase();
    }).collect(Collectors.toList());
}
```

4.16. 测试reduce操作的恒等值：

验证reduce操作需要提供一个恒等值，这个值在流为空的时候会被返回。

```java
@Test
public void testReduceIdentity() {
    int sum = Stream.of(1, 2, 3, 4, 5).reduce(0, Integer::sum);
    assertEquals(15, sum);
    sum = Stream.<Integer>empty().reduce(0, Integer::sum);
    assertEquals(0, sum); // The identity value should be returned for an empty stream
}
```

4.17. 测试peek操作的调试用途：

验证peek操作主要用于调试，它可以让你查看Stream中的值而不改变它们。

```java
@Test
public void testPeekOperation() {
    List<String> list = Stream.of("one", "two", "three")
    .peek(System.out::println)
    .collect(Collectors.toList());
    assertEquals(Arrays.asList("one", "two", "three"), list);
}
```

4.18. 测试flatMap操作的内存占用：

验证flatMap操作可以将多个Stream连接成一个Stream，但是如果每个元素生成的Stream元素非常多，可能会导致内存占用过高。

```java
@Test(expected = OutOfMemoryError.class)
public void testFlatMapMemoryUsage() {
    Stream.iterate(0, i -> i + 1)
    .flatMap(i -> Stream.generate(() -> i).limit(1000000))
    .limit(1000000)
    .collect(Collectors.toList()); // This should throw an OutOfMemoryError
}
```

4.19. 测试Stream.iterate生成大量数据：

验证Stream.iterate可以生成一个无限的Stream，如果你尝试在这个Stream上进行终止操作，如count、collect等，可能会导致程序永不停止，或者内存溢出。

```java
@Test(timeout = 1000)
public void testStreamIterate() {
    long count = Stream.iterate(0, i -> i + 1).limit(1000000).count();
    assertEquals(1000000, count);
}
```

4.20. 测试处理无限Stream：

验证Stream.iterate和Stream.generate可以创建无限的Stream，这些Stream没有固定的大小或结束。如果你对这样的Stream进行count、collect或reduce等需要处理整个Stream的操作，程序可能会进入无限循环。

```java
@Test(expected = OutOfMemoryError.class)
public void testInfiniteStream() {
    Stream.iterate(0, i -> i + 1).collect(Collectors.toList()); // This should throw an OutOfMemoryError
}
```

4.21. 测试reduce操作的使用：

验证reduce操作需要提供一个恒等值（identity），这个值在流为空的时候会被返回，同时它还应该是组合函数的单位元素。

```java
@Test
public void testReduceWithIdentity() {
    int result = Stream.<Integer>empty().reduce(0, Integer::sum);
    assertEquals(0, result);  // The result should be the identity value
}
```

4.22. 测试peek操作的使用：

验证peek操作主要用于调试，它可以让你查看Stream中的值而不改变它们。

```java
@Test
public void testPeekOperation() {
    List<String> list = Stream.of("one", "two", "three")
    .peek(System.out::println)  // This should print the element
    .collect(Collectors.toList());
    assertEquals(Arrays.asList("one", "two", "three"), list);
}
```

4.23. 测试flatMap操作的使用：

验证flatMap操作可以将多个Stream连接成一个Stream，但是如果每个元素生成的Stream元素非常多，可能会导致内存占用过高。

```java
@Test(expected = OutOfMemoryError.class)
public void testFlatMapOperation() {
    Stream.iterate(0, i -> i + 1)
    .flatMap(i -> Stream.generate(() -> i).limit(Integer.MAX_VALUE))
    .limit(Integer.MAX_VALUE)
    .collect(Collectors.toList());  // This should throw an exception
}
```

4.24. 测试Stream.iterate的使用：

验证Stream.iterate可以生成一个无限的Stream，如果你尝试在这个Stream上进行终止操作，如count、collect等，可能会导致程序永不停止，或者内存溢出。

```java
@Test(timeout = 1000)
public void testIterateOperation() {
    long count = Stream.iterate(0, i -> i + 1).limit(10000).count();
    assertEquals(10000, count);
}
```

4.25. 测试Stream的惰性求值特性：

验证Stream的许多操作是惰性的，只有在真正需要结果的时候才会执行。

```java
@Test
public void testLazyEvaluation() {
    AtomicBoolean executed = new AtomicBoolean(false);
    Stream.of("one", "two", "three")
        .filter(e -> {
            executed.set(true);
            return true;
        });
    assertFalse(executed.get());  // The filter operation should not be ex
ecuted
}
```

4.26. 测试boxed类型：

验证IntStream、LongStream和DoubleStream等原始类型Stream比通常的Stream<Integer>、Stream<Long>和Stream<Double>更为高效，因为它们避免了自动装箱和拆箱的开销。

```java
@Test
public void testBoxedStream() {
    List<Integer> list = IntStream.range(0, 1000000).boxed().collect(Collectors.toList());
    assertEquals(1000000, list.size());
}
```

4.27. 测试Stream不是集合：

验证Stream不是数据结构并且不能被重复使用或者并发修改。

```java
@Test(expected = IllegalStateException.class)
public void testStreamIsNotCollection() {
    Stream<String> stream = Stream.of("one", "two", "three");
    stream.count();
    stream.count();  // This should throw an exception
}
```

4.28. 测试使用适当的数据结构：

验证在处理大量数据时，使用适当的数据结构（如数组或集合）可能会比使用Stream更为高效。

```java
@Test
public void testAppropriateDataStructure() {
    long startTime = System.currentTimeMillis();
    List<Integer> list = IntStream.range(0, 10000000).boxed().collect(Collectors.toList());
    long durationWithStream = System.currentTimeMillis() - startTime;

    startTime = System.currentTimeMillis();
    List<Integer> list2 = new ArrayList<>();
    for (int i = 0; i < 10000000; i++) {
        list2.add(i);
    }
    long durationWithLoop = System.currentTimeMillis() - startTime;

    assertTrue(durationWithLoop < durationWithStream);
}
```

4.29. 测试在并行Stream中使用有状态的函数：

验证如果你在并行Stream的map或filter等操作中使用有状态的函数，可能会导致结果不正确。

```java
@Test
public void testStatefulFunctionInParallelStream() {
    List<Integer> list = IntStream.range(0, 10000).parallel()
    .map(new AtomicInteger()::getAndIncrement)
    .boxed().collect(Collectors.toList());
    assertFalse(IntStream.range(0, 10000).allMatch(i -> i == list.get(i)));
  // The order may not be correct due to data race
}
```

4.30. 测试在Stream中使用break或return：

验证在Stream的forEach或其他方法的Lambda表达式中，你不能使用break或return来提前结束处理或跳过某些元素。

```java
@Test
public void testBreakInStream() {
    List<String> list = Stream.of("one", "two", "three")
        .filter(e -> !e.equals("two"))
        .collect(Collectors.toList());
    assertEquals(Arrays.asList("one", "three"), list);
}
```

4.31. 测试collect操作的性能：

验证collect操作可以将Stream的元素收集到一个集合中，但是这需要在内存中存储所有的元素。如果Stream的元素非常多，可能会导致内存溢出。

```java
@Test(expected = OutOfMemoryError.class)
public void testCollectPerformance() {
    Stream.iterate(0, i -> i + 1)
    .limit(Integer.MAX_VALUE)
    .collect(Collectors.toList());  // This should throw an exception
}
```

4.32. 测试在Stream操作中改变输入参数：

在Stream的map、filter等操作中，你应该避免改变输入参数。这些操作应该是纯函数（pure function），即对于相同的输入，总是产生相同的输出，而且不产生任何副作用。

```java
@Test
public void testChangingInputInStream() {
    List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3));
    list.stream().map(e -> e = e + 1).collect(Collectors.toList());
    assertEquals(Arrays.asList(1, 2, 3), list);  // The original list should not be changed
}
```

4.33. 测试在并行Stream中使用线程不安全的数据结构：

如果你在并行Stream的map或collect等操作中使用线程不安全的数据结构，可能会导致数据竞争和不一致的结果。

```java
@Test
public void testUnsafeParallelStream() {
    List<Integer> list = new ArrayList<>();
    IntStream.range(0, 10000).parallel().forEach(list::add);
    assertNotEquals(10000, list.size());  // The size may not be 10000 due to data race
}
```

4.34. 测试在Stream中使用null值：

在Stream中使用null值可能会抛出NullPointerException。

```java
@Test(expected = NullPointerException.class)
public void testNullInStream() {
    Stream.of("one", null, "three").map(String::length).collect(Collectors.toList());  // This should throw an exception
}
```

4.35. 测试Optional的使用：

findFirst、findAny等返回的是Optional，而不是实际的值。

```java
@Test
public void testOptionalInStream() {
    Stream<String> stream = Stream.of("one", "two", "three");
    Optional<String> optional = stream.findFirst();
    assertTrue(optional.isPresent());
    assertEquals("one", optional.get());
}
```