《Java 8函数式编程》

图书基本信息

书名：《Java 8函数式编程》

13位ISBN编号：9787115384886

出版时间：2015-3

作者：[英] Richard Warburton

页数：148

译者：王群锋

《Java 8函数式编程》

内容概要

通过每一章的练习快速掌握Java 8中的Lambda表达式
分析流、高级集合和其他Java 8类库的改进
利用多核CPU提高数据并发的性能
将现有代码库和库代码Lambda化
学习Lambda表达式单元测试和调试的实践解决方案
用Lambda表达式实现面向对象编程的SOLID原则
编写能有效执行消息传送和非阻塞I/O的并发应用

《Java 8函数式编程》

作者简介

作者简介：
Richard Warburton
一位经验丰富的技术专家，善于解决复杂深奥的技术问题，拥有华威大学计算机科学专业博士学位。近期他一直从事高性能计算方面的数据分析工作。他是英国伦敦Java社区的领导者，组织过面向Java 8中Lambda表达式、日期和时间的Adopt-a-JSR项目，以及Openjdk Hackdays活动。Richard还是知名的会议演讲嘉宾，曾在JavaOne、DevoxxUK和JAX London等会议上演讲。
译者简介：
王群锋
毕业于西安电子科技大学，现任职于IBM西安研发中心，从事下一代统计预测软件的开发运维工作。

《Java 8函数式编程》

书籍目录

《Java 8函数式编程》

精彩短评

1、随便看看，没啥用
2、书不错，简短易读，翻译的也没毛病，配图也比较用心。最近RxJava挺热门的嘛。
3、几乎可以想象oracle的研究员们在设计Java8时候如何费煞心思地既保持向后兼容、又带来质的飞跃
4、不全面，不过足够让人再次爱上Java
5、特别惊诧于Java中引入函数式编程的方法，这么天然无邪，和原有的API又整合得毫无痕迹
6、很不错的一本介绍函数式编程的书，深入浅出，后面还有一些DSL和NIO的内容。
7、还可以，主讲函数式方面的。
8、入门，想要精通还得细细揣摩
9、比较简短，速度入门，然后在工作中深入。
10、挺好的一本书，通俗易懂，简洁明了。用这本书入门函数式编程挺好
11、快速了解Java 8 Lambda 和stream
12、入门的几章还好,后面有点吃力,源码要吃透不容易...
13、灰常好，不仅介绍了新增的lambda，对函数式编程思维也有阐述，最后函数式编程对设计模式的简化部分令人很有启发！
14、很不错的一本书，学习Java8的新知识以及Java的函数式编程最佳书籍
15、能够对Java8中的lambda表达式有个初步的了解
16、这本书除了介绍函数式编程的使用方法外，还涉及单元测试，重构，设计原则和设计方法，但都是蜻蜓点水式的过了一遍。Java的函数式编程并不纯粹，也不够完美，但至少给广大Java程序员带来一种新的尝试。
17、技术书要看纸质版，还要做练习。
18、函数式编程入门推荐书籍
19、回头得把习题做一下！
20、入门挺不错的
21、对于java8中最重要的新特性lambda有一个初步的了解，明白他能做什么，不能做什么！还是非常值得读的。
22、内容实用，适合想上手使用Java8新特性的人。最后两章有点在拼凑内容的感觉
23、会帮助理解java8中函数式编程的理解，但感觉很多的是在教你怎么用，而对于背后是如何实现的，是语法层面的还是实现层面的，编译器在编译时是如何判断和分派的等等，没有很好的涉及。比如，对于函数的引用时，没有()和有()在编译和实现上是一种什么样的区别...
24、篇幅不长，大致介绍了Java 8引入的Stream API和lambda表达式的用法，对以往Java项目代码可做的改进等。还包括vertx和RxJava的简要介绍。阅读本书之外还需要更多的实践。
25、对函数式编程倒是讲的很详细
26、不错的Java8新特性入门书，例子比较丰富。不过篇幅小，并没有介绍太深入，偏实用。
27、凑合吧 介绍了一些新特性 但感觉不够深入
28、大部分内容在飞机上看完了 还不错 比较薄 内容简明
29、入门还不错
30、java 8 函数式编程入门：）
31、比较薄，不过关于JAVA8讲的不错，快速入门，比较精简，没有啥废话。要是增加课后习题就给5星了
32、一本书能这样深入浅出将java8新特性的来龙去脉讲清楚，虽然贵点也值了，好评
33、比较简短，随便看看
34、看过最好的Java 8函数式编程资料，即系统全面又实例生动，值得再读一遍的好书。
35、基本点都涵盖到了，连rx、completablefuture也有所讲解，对于用java进行函数式编程是个不错的入门书籍，但是想要掌握还是得多多练习。
36、读第一遍可能会开始尝试map-reduce之类的函数式方法，当然还有lambda表达式，然后后面写东西的时候有些东西想不起来就拿来翻一翻，每次总能学到一些东西，这本书可以当作入门，当然也可以当作参考书，放在手边，多翻一翻。

37、深入浅出

《Java 8函数式编程》

章节试读

1、《Java 8函数式编程》的笔记-第55页

在collect方法中使用partitioningBy方法进行分块。
通过判断true或false，将数据分成两块。
List<String> names = Arrays.asList("Jack", "Linda", "Mark", "JJ", "M");
Map<Boolean, List<String>> partitionMap = names.stream()
.collect(Collectors.partitioningBy(s -> s.length() > 3));
System.out.println("partitionMap: " + partitionMap);
运行结果：
partitionMap: {false=[JJ, M], true=[Jack, Linda, Mark]}

2、《Java 8函数式编程》的笔记-第13页

2.ThreadLocal Lambda表达式。Java有一个ThreadLocal类，作为容器保存了当前线程里局部变量的值。Java 8为该类新加了一个工厂方法，接受一个Lambda表达式，并产生一个新的ThreadLocal对象，而不用使用继承，语法上更加简洁。

a. 在Javadoc或集成开发环境（IDE）里找出该方法。
b. DateFormatter类是非线程安全的。使用构造函数创建一个线程安全的DateFormatter对象，并输出日期，如“ 01-Jan-1970 ”。

ThreadLocal<DateFormatter> threadLocal = ThreadLocal
.withInitial(() -> new DateFormatter(new SimpleDateFormat("dd-MMM-yyyy")));
System.out.println(threadLocal.get().valueToString(new Date()));

3、《Java 8函数式编程》的笔记-第56页

在collect方法中使用groupingBy方法进行分组。
其实数据分块更像是数据分组的子集，如下用groupingBy实现与partitioningBy一样效果的代码。
Map<Boolean, List<String>> partitionMap = names.stream()
.collect(Collectors.partitioningBy(s -> s.length() > 3));
System.out.println("partitionMap: " + partitionMap);

Map<Boolean, List<String>> partitionMap2 = names.stream()
.collect(Collectors.groupingBy(s -> s.length() > 3));
System.out.println("partitionMap2: " + partitionMap2);
数据分组更加通用。

Map<Character, List<String>> groupingMap = names.stream()
.collect(Collectors.groupingBy(s -> s.charAt(0)));
System.out.println("groupingMap: " + groupingMap);通过首字符进行分组，首字符作为Key。

4、《Java 8函数式编程》的笔记-第90页

1. 重构遗留代码时考虑如何使用Lambda表达式，有一些通用的模式。
2. 如果想要对复杂一点的Lambda表达式编写单元测试，将其抽取成一个常规的方法。

3. peek方法能记录中间值，在调试时非常有用。

5、《Java 8函数式编程》的笔记-第58页

1. 先对一系列字符串进行分组，分组的结果是key（字符串首字母）: value（字符串）
2. 对每个key对应的value计算个数，返回结果是key（字符串首字母）: value（个数）

一般完成上述操作需要分成两次，但是在collect方法中使用groupingBy的两个参数的方法就可以很直接的解决这个问题。

```
@Test
public void test5() {
    List&lt;String&gt; names = Arrays.asList("Jack", "Linda", "Mark", "JJ", "M");
    Map&lt;String, Long&gt; map = lengthOfName(names);
    System.out.println("map: " + map);
}

// 先把首字符相同的字符串放进同个map,首字符作为key,
// 然后计算每个key对应的有多少个String对象，并把计算出来的值作为value
public Map&lt;String, Long&gt; lengthOfName(List&lt;String&gt; names) {
    return names.stream()
      .collect(Collectors.groupingBy(s -&gt; String.valueOf(s.charAt(0)),
    Collectors.mapping(s -&gt; s, Collectors.counting())));
}
```

运行结果：

```
map: {J=2, L=1, M=2}
```

其实，可以对counting方法进行各种替换，如替换成toList方法，那么返回值就变成了Map<String, List<String>>。

6、《Java 8函数式编程》的笔记-第48页

使用Optional对象有两个目的：首先，Optional对象鼓励程序员适时检查变量是否为空，以避免代码缺陷；其次，它将一个类的API中可能为空的值文档化，这比阅读实现代码要简单很多。

```
@Test
public void testOptional() {
    Optional&lt;String&gt; a = Optional.of("a");
    Assert.assertEquals("a", a.get());

    Optional emptyOptional = Optional.empty();
    Optional alsoEmpty = Optional.ofNullable(null);
    Assert.assertFalse(emptyOptional.isPresent());

    Assert.assertTrue(a.isPresent());

    Assert.assertEquals("b", emptyOptional.orElse("b"));

    Assert.assertEquals("c", emptyOptional.orElseGet(() -&gt; "c"));
```

```
    System.out.println("emptyOptional: " + emptyOptional.get());//空值不能获取，会
抛NoSuchElementException错误
}
```

7、《Java 8函数式编程》的笔记-第75页

影响并行流性能的主要因素有5个。
1. 数据块大小输入数据的大小会影响并行化处理对性能的提升。将问题分解之后并行化处理，再将结果合并会带来额外的开销。因此只有数据足够大、每个数据处理管道花费的时间足够多时，并行化处理才有意义。
2. 源数据结构每个管道的操作都基于一些初始数据源，通常是集合。将不同的数据源分割相对容易，这里的开销影响了在管道中并行处理数据时到底能带来多少性能上的提升。
3. 装箱处理基本类型比处理装箱类型要快。
4.核的数量极端情况下，只有一个核，因此完全没必要并行化。显然，拥有的核越多，获得潜在性能提升的幅度就越大。在实践中，核的数量不单指你的机器上有多少核，更是指运行时你的机器能使用多少核。这也就是说同时运行的其他进程，或者线程关联性（强制线程在某些核或CPU上运行）会影响性能。
5. 单元处理开销比如数据大小，这是一场并行执行花费时间和分解合并操作开销之间的战争。花在流中每个元素身上的时间越长，并行操作带来的性能提升越明显。
根据性能好坏，将核心类库提供的通用数据结构分成一下3组：
1. 性能好ArrayList、数组或IntStream.range，这些数据结构支持随机读取，也就是说它们能轻而易举被任意分解。
2. 性能一般HashSet、TreeSet，这些数据结构不易公平地被分解，但是大多数时候分解是可能的。
3. 性能差有些数据结构难于分解，比如，可能要花O(N)的时间复杂度来分解问题。其中包括LinkedList，对半分解太难了。还有Stream.iterate和BufferedReader.lines，它们长度未知，因此很难预测该在哪里分解。
在流中单独操作每一块的种类时，可以分成两种不同的操作：无状态的和有状态的。无状态操作整个过程中不必维护状态，有状态操作则有维护状态所需的开销和限制。无状态操作包括map、filter和flatMap，有状态操作包括sorted、distinct和limit。

8、《Java 8函数式编程》的笔记-第31页

本章介绍的概念能够帮助用户写出更简单的代码，因为这些概念描述了数据上的操作，明确了要达成什么转化，而不是说明如何转化。这种方式写出的代码，潜在的缺陷更少，更直接地表达了程序员的意图。明确要达成什么转化，而不是说明如何转化的另外一层含义在于写出的函数没有副作用。
没有副作用的函数不会改变程序或外界的状态。
Lambda表达式中向控制台输出信息或给局部变量赋值都是有副作用的。
无论何时，将Lambda表达式传给Stream上的高阶函数，都应该尽量避免副作用。唯一的例外是forEach方法，它是一个终结方法。

9、《Java 8函数式编程》的笔记-第66页

在collect方法中使用reducing方法，写法恶心，而且效率底下，so 暂时不研究。

10、《Java 8函数式编程》的笔记-第44页

类中重写的方法优先级高于接口中定义的默认方法。简言之，类中重写的方法胜出。这样的设计主要是由默认方法的目的决定的，增加默认方法的目的是为了在接口上向后兼容。让类中重写方法的优先级高于默认方法能简化很多继承问题。

11、《Java 8函数式编程》的笔记-第33页

1.只用reduce和Lambda表达式写出实现Stream上的map操作的代码，如果不想返回Stream，可以返回一个List。

```
@Test
public void test1() {
    List&lt;String&gt; names = Arrays.asList("Jack", "Linda", "Mark", "JJ", "M");
    List&lt;Integer&gt; lengths = likeMap(names, s -&gt; s.length());
    System.out.println("lengths: " + lengths);
}

public &lt;T, R&gt; List&lt;R&gt; likeMap(List&lt;T&gt; resource, Function&lt;T, R&gt; function) {
    return resource.stream().reduce(new ArrayList&lt;R&gt;(), (acc, x) -&gt; {
        List&lt;R&gt; newAcc = new ArrayList&lt;&gt;(acc);
        newAcc.add(function.apply(x));
        return newAcc;
    }, (List&lt;R&gt; a, List&lt;R&gt; b) -&gt; {
        List&lt;R&gt; left = new ArrayList&lt;&gt;(a);
        left.addAll(b);
        return left;
    });
}
```

2.只用reduce和Lambda表达式写出实现Stream上的filter操作的代码，如果不想返回Stream，可以返回一个List。

```
@Test
public void test2() {
    List&lt;String&gt; names = Arrays.asList("Jack", "Linda", "Mark", "JJ", "M");
    //过滤不含'a'字符的字符串
    List&lt;String&gt; result = likeFilter(names, s -&gt; s.contains("a"));
    System.out.println("result: " + result);
}

public &lt;T&gt; List&lt;T&gt; likeFilter(List&lt;T&gt; resorce, Predicate&lt;? super T&gt; predicate) {
    return resorce.stream().reduce(new ArrayList&lt;T&gt;(), (acc, x) -&gt; {
        List&lt;T&gt; newAcc = new ArrayList&lt;&gt;(acc);
        if (predicate.test(x))
            newAcc.add(x);
        return newAcc;
    }, (List&lt;T&gt; left, List&lt;T&gt; right) -&gt; {
        List&lt;T&gt; list = new ArrayList&lt;&gt;(left);
        list.addAll(right);
        return list;
    });
}
```

12、《Java 8函数式编程》的笔记-第69页

并发：在一个CPU上，一个时间段，执行多个任务，利用不断切换任务让人误以为同时执行。

并行：在多个CPU上，每个CPU上分别执行自己的任务，这是真正意义上的同时。
数据并行化是指将数据分成块，为每块数据分配单独的处理单元。
任务并行化相当于多线程。

13、《Java 8函数式编程》的笔记-第36页

Java 8提供了类型转换的函数接口。

ToIntFunction接口：
@FunctionalInterface
public interface ToIntFunction<T> {
  int applyAsInt(T value);
}
IntFunction接口：
@FunctionalInterface
public interface IntFunction<R> {
  R apply(int value);
}
在Java 8中，仅对整型、长整型和双浮点型做了特殊处理。

```
@Test
public void test4_4() {
   List&amp;lt;Integer&amp;gt; ints = Arrays.asList(1, 2, 3, 4, 5);
   System.out.println(ints.stream()
         .mapToInt(num-&amp;gt;num+1).summaryStatistics().getMax());
}
```

其中mapToInt方法中的参数就是ToIntFunction接口。

14、《Java 8函数式编程》的笔记-第148页

15、《Java 8函数式编程》的笔记-第67页

并没有认真做。

16、《Java 8函数式编程》的笔记-第31页

高阶函数是指接受另外一个函数作为参数，或返回一个函数的函数。如果函数的参数列表里包含函数接口，或该函数返回一个函数接口，那么该函数就是高阶函数。
Stream接口中几乎所有函数都是高阶函数。

17、《Java 8函数式编程》的笔记-第61页

将一串字符串串在一起，原本在collect方法中使用joining方法分分钟解决，但是为了练习，使用reduce方法解决。

```
@Test
```

```
public void testStringCombiner() {
    List&lt;String&gt; names = Arrays.asList("Jack", "Linda", "Mark", "JJ", "M");
    StringCombiner combined = names.stream()
            .reduce(new StringCombiner(", ", "[ ", " ]"),
                    StringCombiner::add,
                    StringCombiner::merge);
    String result = combined.toString();
    System.out.println("result: " + result);
}public class StringCombiner {

    private final String delim;
    private final String prefix;
    private final String suffix;
    private final StringBuilder builder;

    public StringCombiner(String delim, String prefix, String suffix) {
        this.delim = delim;
        this.prefix = prefix;
        this.suffix = suffix;
        builder = new StringBuilder();
    }

    // BEGIN add
    public StringCombiner add(String element) {
        if (areAtStart()) {
            builder.append(prefix);
        } else {
            builder.append(delim);
        }
        builder.append(element);
        return this;
    }
    // END add

    private boolean areAtStart() {
        return builder.length() == 0;
    }

    // BEGIN merge
    public StringCombiner merge(StringCombiner other) {
        if (other.builder.length() &gt; 0) {
            if (areAtStart()) {
                builder.append(prefix);
            } else {
                builder.append(delim);
            }
            builder.append(other.builder, prefix.length(), other.builder.length());
        }
        return this;
```

```
    }
    // END merge

    @Override
    public String toString() {
        if (areAtStart()) {
            builder.append(prefix);
        }
        builder.append(suffix);
        return builder.toString();
    }

}
```

三个参数的reduce方法，第二个和第三个参数都是接收两个参数的方法，但是StringCombiner对象的add方法和merge方法都是接受一个参数的对象，那另一个参数去哪了？调用方法的对象就是第一个参数。

所以，StringCombiner::add的两个参数分别是this和name，StringCombiner::merge同样如此。

定制收集器（难点、并未掌握）

```
import java.util.Collections;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;

public class StringCollector implements Collector&lt;String, StringCombiner, String&gt; {
    private static final Set&lt;Characteristics&gt; characteristics = Collections.emptySet();

    private String delim;
    private String prefix;
    private String suffix;

    public StringCollector() {
        super();
    }

    public StringCollector(String delim, String prefix, String suffix) {
        super();
        this.delim = delim;
        this.prefix = prefix;
        this.suffix = suffix;
    }

    @Override
    public Supplier&lt;StringCombiner&gt; supplier() {
```

```
    return () -&gt; new StringCombiner(delim, prefix, suffix);
  }

  @Override
  public BiConsumer&lt;StringCombiner, String&gt; accumulator() {
    return StringCombiner::add;
  }

  @Override
  public BinaryOperator&lt;StringCombiner&gt; combiner() {
    return StringCombiner::merge;
  }

  @Override
  public Function&lt;StringCombiner, String&gt; finisher() {
    return StringCombiner::toString;
  }

  @Override
  public Set&lt;java.util.stream.Collector.Characteristics&gt; characteristics() {
    return characteristics;
  }
```

}首先要实现Collector接口，supplier方法用来创建对象，accumulator方法用来添加当前元素进对象里，combiner方法用来合并两个对象，finisher方法返回收集操作的最终结果。

Java 8 有一个java.util.StringJoiner类，它的作用和StringCombiner一样，有类似的API。

18、《Java 8函数式编程》的笔记-第55页

```
List&lt;String&gt; names = Arrays.asList("Jack", "Linda", "Mark", "JJ", "M");

Optional&lt;String&gt; maxLength1 = names.stream()
    .collect(Collectors.maxBy(Comparator.comparing(String::length)));
System.out.println("maxLength1: " + maxLength1);

Optional&lt;String&gt; maxLength2 = names.stream().
    max(Comparator.comparing(String::length));
System.out.println("maxLength2: " + maxLength2);
```

收集器collect方法中使用maxBy方法，但是，与直接使用max方法的效果完全一样。

```
Double avgLength = names.stream()
    .collect(Collectors.averagingInt(String::length));
System.out.println("avgLength: " + avgLength);
```

收集器collect方法中使用averagingInt方法计算平均值，得到一个Double数值。

19、《Java 8函数式编程》的笔记-第78页

数组上的并行化操作1. parallelPrefix 任意给定一个函数，计算数组的和
2. parallelSetAll 使用Lambda表达式更新数组元素
3. parallelSort 并行化对数组元素排序
例6-8 使用并行化数组操作初始化数组

```
@Test
public void test6_8() {
    double[] values = parallelInitialize(10);
    Arrays.stream(values).forEach(System.out::println);
}

public static double[] parallelInitialize(int size) {
    double[] values = new double[size];
    Arrays.parallelSetAll(values, i -&gt; i + 0.1);
    return values;
}
```

20、《Java 8函数式编程》的笔记-第10页

例 2-10 使用菱形操作符，根据方法签名做推断

```
userHashmap(new HashMap&lt;&gt;());
```

...

private void userHashmap(Map<String, String> values);Java 8中对类型推断系统的改善值得一提。上面的例子将new HashMap<>()传给userHashmap方法，即使编译器拥有足够的信息，也无法在Java 7中通过编译。
例 2-12 Predicate接口的源码，接受一个对象，返回一个布尔值

```
public interface Predicate&lt;T&gt; {
    boolean test(T t);
}
```

BinaryOperator接口，需要接受一个泛型，若没有写明泛型，则默认为Object。

```
@FunctionalInterface
public interface BinaryOperator&lt;T&gt; extends BiFunction&lt;T,T,T&gt; {
   /**
    * Returns a {@link BinaryOperator} which returns the lesser of two elements
    * according to the specified {@code Comparator}.
    *
    * @param &lt;T&gt; the type of the input arguments of the comparator
    * @param comparator a {@code Comparator} for comparing the two values
    * @return a {@code BinaryOperator} which returns the lesser of its operands,
    *         according to the supplied {@code Comparator}
    * @throws NullPointerException if the argument is null
    */
   public static &lt;T&gt; BinaryOperator&lt;T&gt; minBy(Comparator&lt;? super T&gt; comparator) {
      Objects.requireNonNull(comparator);
      return (a, b) -&gt; comparator.compare(a, b) &lt;= 0 ? a : b;
   }
```

```
  /**
   * Returns a {@link BinaryOperator} which returns the greater of two elements
   * according to the specified {@code Comparator}.
   *
   * @param &lt;T&gt; the type of the input arguments of the comparator
   * @param comparator a {@code Comparator} for comparing the two values
   * @return a {@code BinaryOperator} which returns the greater of its operands,
   *         according to the supplied {@code Comparator}
   * @throws NullPointerException if the argument is null
   */
  public static &lt;T&gt; BinaryOperator&lt;T&gt; maxBy(Comparator&lt;? super T&gt; comparator) {
    Objects.requireNonNull(comparator);
    return (a, b) -&gt; comparator.compare(a, b) &gt;= 0 ? a : b;
  }
}
```

21、《Java 8函数式编程》的笔记-第45页

其实应该称作多重实现。

例4-19 Jukeboxpublic interface Jukebox {
    public default String rock() {
        reutrn "... all over the world!";
    }
}
例4-20 Carriagepublic interface Carriage {
    public default String rock() {
        return "... from side to side";
    }
}
例4-21 实现rock方法public class MusicalCarriage implements Carriage, Jukebox {
    @Override
    public String rock() {
        return Carriage.super.rock();
    }
}
多重实现，需要重写默认方法，也可以通过super关键字指明使用哪个接口的默认方法。

22、《Java 8函数式编程》的笔记-第40页

该注解会强制javac检查一个接口是否符合函数接口的标准。如果该注释添加给一个枚举类型、类或另一个注释，或者接口包含不止一个抽象方法，javac就会报错。
也可以看出，被@FunctionalInterface注解的接口也可以有静态方法或default方法。

23、《Java 8函数式编程》的笔记-第75页

要使用并行化运算必须遵守一些规则和限制。

reduce三个参数的方法中，如果是串行化运算就不会出现问题，但是并行化运算会出现初始值和每个元素进行运算。

```
@Test
public void testLimit() {
    List&lt;Integer&gt; numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
    int result1 = numbers.stream()
            .reduce(2, (acc, x) -&gt; acc + x, (a, b) -&gt; a + b);
    System.out.println("result1: " + result1);

    int result2 = numbers.stream().parallel()
            .reduce(2, (acc, x) -&gt; acc + x, (a, b) -&gt; a + b);
    System.out.println("result2: " + result2);
    }
```

运算结果：

```
result1: 47
result2: 63
```

并行运算每个元素都与初始值计算了一次，所以，要进行并行运算，初始值必须为0。

parallel方法是将流转换成并行流，
sequential方法是将流转化成串行流。如果同时调用了parallel和sequential方法，最后调用的那个方法起效。

24、《Java 8函数式编程》的笔记-第8页

Lambda表达式引用的是值，而不是变量。如果Lambda表达式引用表达式外部的变量，那这个变量只能是一个既成事实上的final变量。

```
String name = getUserName();
name = formatUserName(name);
button.addActionListener(event -&gt; System.out.println("hi " + name));
```

以上代码无法通过编译，name可以不用final声明，但是必须和final一样，只能被赋一次值。

25、《Java 8函数式编程》的笔记-第9页

函数接口是只有一个抽象方法的接口，用作Lambda表达式的类型。这就是函数接口，接口中单一方法的命名并不重要，只要方法签名和Lambda表达式的类型匹配即可。

26、《Java 8函数式编程》的笔记-第1页

Q: 为什么要会有Java 8？
A: 适应现代多核CPU，让代码在多核CPU上高效运行。
面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象。以前传递行为的方式是传递拥有具有某种行为的对象，Java 8有了Lambda表达式和方法引用（因为Lambda表达式和方法引用都是直接表述一种行为）后，传递行为的方式更加直观。

27、《Java 8函数式编程》的笔记-第54页

例5-5 使用toCollection，用定制的集合收集元素
stream.collect(Collectors.toCollection(TreeSet::new));

28、《Java 8函数式编程》的笔记-第80页

2. 例6-11中的代码把列表中的数字相乘，然后再将所得结果乘以5。顺序执行这段程序没有问题，但并行执行时有一个缺陷，使用流并行化执行该段代码，并修复缺陷。

例6-11 把列表中的数字相乘，然后再将所得结果乘以5，该实现有一个缺陷// 串行reduce初始值为任意值并不会有问题
public static int multiplyThrough(List<Integer> linkedListOfNumbers) {
return linkedListOfNumbers.stream().reduce(5, (acc, x) -> x * acc);
}@Test
public void test6_9_2() {
List<Integer> linkedListOfNumbers = Arrays.asList(1, 2, 3, 4, 5);
System.out.println("multiplyThrough: " + multiplyThrough(linkedListOfNumbers));
System.out.println("serialMultiplyThrough: " + serialMultiplyThrough(linkedListOfNumbers));
}

// 并行reduce初始值必须为特定值，所以为了保持结果不变，把最终结果 * 5
public static int serialMultiplyThrough(List<Integer> linkedListOfNumbers) {
return 5 * linkedListOfNumbers.parallelStream().reduce(1, (acc, x) -> x * acc);
}
3.例6-12中的代码计算列表中数字的平方和。尝试改进代码性能，但不得牺牲代码质量。只需要一些简单的改动即可。
例6-12 求列表元素的平方和，该实现方式性能不高
public int solwSumOfSquares(List<Integer> linkedListOfNumbers) {
return linkedListOfNumbers.parallelStream().map(x -> x * x)
.reduce(0, (acc, x) -> acc + x);
}@Test
public void test6_9_3() {
List<Integer> linkedListOfNumbers = Arrays.asList(1, 2, 3, 4, 5);
System.out.println("solwSumOfSquares: " + solwSumOfSquares(linkedListOfNumbers));
System.out.println("fastSumOfSquares: " + fastSumOfSquares(linkedListOfNumbers));
System.out.println("serialFastSumOfSquares: " + serialFastSumOfSquares(linkedListOfNumbers));
}

// 高效并行计算
public int fastSumOfSquares(List<Integer> linkedListOfNumbers) {
return linkedListOfNumbers.parallelStream().mapToInt(x -> x * x).sum();
}

// 高效串行计算
public int serialFastSumOfSquares(List<Integer> linkedListOfNumbers) {
return linkedListOfNumbers.stream().mapToInt(x -> x * x).sum();
}

29、《Java 8函数式编程》的笔记-第66页

新增一些对Map对象操作的方法。

```
@Test
public void test5_4() {
    Map&lt;String, String&gt; map = new HashMap&lt;String, String&gt;(){{
        put("J", "Jack");
        put("L", "Linda");
        put("M", "Mark");
    }};

    String str1 = map.computeIfAbsent("A", n -&gt; n + "_Article");
    System.out.println("str1: " + str1);

    String str2 = map.compute("J", (n, m) -&gt; n.toString() + "_" + m.toString());
                                // n和m分别是key和value
    System.out.println("str2: " + str2);
}
```

运行结果：

```
str1: A_Article
str2: J_Jack
```

对Map对象的迭代，用key和value同时迭代。

```
Map&lt;String, Integer&gt; resultMap = new HashMap&lt;&gt;();
map.forEach((key, value)-&gt;{
    resultMap.put(key, value.length());
});
System.out.println("resultMap: " + resultMap);运行结果：
resultMap: {A=9, J=6, L=5, M=4}
```

30、《Java 8函数式编程》的笔记-第52页

在一个有序集合中创建一个流时，流中的元素就按出现顺序排列。如果集合本身就是无序的，由此生成的流也是无序的。
List是有序的，HashSet是无序的。
一些操作在有序的流上开销更大，调用unordered方法消除这种顺序就能解决该问题。大多数操作都是在有序流上效率更高，比如filter、map和reduce等。
forEach不能保证顺序处理，forEachOrdered能够保证顺序处理。

31、《Java 8函数式编程》的笔记-第19页

3.3.1 collect(toList())

```
List&lt;String&gt; collected = Stream.of("a", "b", "c").collect(Collectors.toList());
```

3.3.2 map如果有一个函数可以将一种类型的值转换成另外一种类型，map操作就可以使用该函数，将一个流中的值转换成一个新的流。

```
List&lt;String&gt; strList = Arrays.asList("str1", "str2", "str3", "str4", "str5", "str6");
List&lt;Integer&gt; intList = strList.stream()
    .map(s -&gt; Integer.parseInt(String.valueOf(s.charAt(3))))
```

```
    .collect(Collectors.toList());
```

3.3.4 flapMapflatMap方法可用Stream替换值，然后将多个Stream连接成一个Stream。

```
List&lt;Integer&gt; integerResult = Stream.of(Arrays.asList(1, 2), Arrays.asList(3, 4))
            .flatMap(n -&gt; n.stream())
            .collect(Collectors.toList());
      System.out.println("integerResult: " + integerResult);
```

相当于下面代码：

```
Stream&lt;List&lt;Integer&gt;&gt; together = Stream.of(Arrays.asList(1, 2), Arrays.asList(3, 4));
Stream&lt;Integer&gt; integers = together.flatMap(n -&gt; n.stream());
List&lt;Integer&gt; integerList = integers.collect(Collectors.toList());
System.out.println("integerList: " + integerList);
```

flatMap中的方法返回的是Stream类型，也就是把所有返回的Stream对象全部串成一个Stream对象作为结果。

3.3.5 max和min

```
List&lt;String&gt; articles = Arrays.asList("Jack", "Linda", "Mark", "JJ", "M");
Optional&lt;String&gt; minOption = articles.stream()
    .min(Comparator.comparing(a -&gt; a.length()));
String minStr = minOption.get();
System.out.println("minStr: " + minStr);
```

comparing方法接受一个函数并返回一个函数。

3.3.7 reduce　——重点、难点

reduce操作可以实现从一组值中生成一个值。
count()、min()和max()方法都是reduce操作，因其常用而被纳入标准库中。

```
int count_1 = Stream.of(1, 2, 3, 4, 5)
   .reduce(0, (acc, name) -&gt; acc + name);
System.out.println("count_1: " + count_1);T reduce(T identity, BinaryOperator&lt;T&gt; accumulator);
```

以下为BinaryOperator接口源码：

```
@FunctionalInterface
public interface BinaryOperator&lt;T&gt; extends BiFunction&lt;T,T,T&gt; {
  public static &lt;T&gt; BinaryOperator&lt;T&gt; minBy(Comparator&lt;? super T&gt; comparator) {
    Objects.requireNonNull(comparator);
    return (a, b) -&gt; comparator.compare(a, b) &lt;= 0 ? a : b;
  }
  public static &lt;T&gt; BinaryOperator&lt;T&gt; maxBy(Comparator&lt;? super T&gt; comparator) {
    Objects.requireNonNull(comparator);
    return (a, b) -&gt; comparator.compare(a, b) &gt;= 0 ? a : b;
  }
}
```

BinaryOperator接口并没有抽象方法，所以它的抽象方法只能在其父类或父类的父类中。
以下为BiFunction接口源码：

```
@FunctionalInterface
public interface BiFunction&lt;T, U, R&gt; {
  R apply(T t, U u);

  default &lt;V&gt; BiFunction&lt;T, U, V&gt; andThen(Function&lt;? super R, ? extends V&gt; after) {
    Objects.requireNonNull(after);
```

```
        return (T t, U u) -&gt; after.apply(apply(t, u));
    }
}
```
所以 BinaryOperator<T>接口实现的是T apply(T t, T u);方法，因此，当reduce方法的参数为两个时，操作的只能都是同类型的参数，返回同类型的值。

```
int count_2 = Stream.of("Jack", "Linda", "Mark", "JJ", "M")
    .reduce(0, (acc, s) -&gt; acc + s.length(), (a, b) -&gt; a + b);
System.out.println("count_2: " + count_2);
```
reduce方法的参数为三个时<U> U reduce(U identity,
BiFunction<U, ? super T, U> accumulator,
BinaryOperator<U> combiner);所以BiFunction<U, ? super T, U>接口实现的是U apply(U u, ? super T t);方法，其中T类型是Stream<T>中的T，所以(acc, s)中可以指导s是String类型。

32、《Java 8函数式编程》的笔记-第51页

```
String[]::new
```
以上String[]类型也行，具体怎么用之后再考虑。

33、《Java 8函数式编程》的笔记-第57页

在collect方法中使用joining方法可以将流中的值串成一个字符串。

```
String friends = names.stream()
    .collect(Collectors.joining(", ", "[ ", " ]"));
System.out.println("friends: " + friends);
```
运行结果：
```
friends: [ Jack, Linda, Mark, JJ, M ]
```

34、《Java 8函数式编程》的笔记-第73页

例6-3 使用蒙特卡洛模拟法并行化模拟掷骰子事件

```
@Test
public void testParallelDiceRolls() {
    double fraction = 1.0 / N;
    long start1 = System.currentTimeMillis();
    Map&lt;Integer, Double&gt; resultMap1 = IntStream
            .range(0, N)
            .parallel()
            .mapToObj(twoDiceThrows())
            .collect(Collectors
                    .groupingBy(side-&gt;side,
                        Collectors.summingDouble(n -&gt; fraction)));
    System.out.println("resultMap1: " + resultMap1);
    long time1 = System.currentTimeMillis() - start1;
```

```
    System.out.println("time1: " + (time1 * 1.0 / 1000) + " s");

    long start2 = System.currentTimeMillis();
    Map&lt;Integer, Double&gt; resultMap2 = IntStream
            .range(0, N)
            .mapToObj(twoDiceThrows())
            .collect(Collectors
                    .groupingBy(side-&gt;side,
                        Collectors.summingDouble(n -&gt; fraction)));
    System.out.println("resultMap2: " + resultMap2);
    long time2 = System.currentTimeMillis() - start2;
    System.out.println("time2: " + (time2 * 1.0 / 1000) + " s");
}

private IntFunction&lt;Integer&gt; twoDiceThrows() {
    return i -&gt; {
        ThreadLocalRandom random = ThreadLocalRandom.current();
        int firstThrow = random.nextInt(1, 7);
        int secondThrow = random.nextInt(1, 7);
        return firstThrow + secondThrow;
    };
}运行结果：
resultMap1: {2=0.02780246, 3=0.05556867, 4=0.08336466000000001, 5=0.11114840999999999,
6=0.13888492000000002, 7=0.16660306000000003, 8=0.13885089999999997, 9=0.11113914999999999,
10=0.08334594999999999, 11=0.05550833999999996, 12=0.02778348}
time1: 2.098 s
resultMap2: {2=0.02775076, 3=0.05560599, 4=0.08328941000000001, 5=0.11112538, 6=0.13884923,
7=0.16664901000000001, 8=0.13892618, 9=0.11111408, 10=0.08337263, 11=0.05553474, 12=0.02778259}
time2: 6.11 s
```

当数据量足够大时，并行化计算可以提升相当多的速度，反之，数据量不足时，并行化计算不仅没有优势，速度还会比较慢。

《Java 8函数式编程》

版权说明