Java 8 之 Stream 学习笔记整理

作者：汪文君

QQ：532500648

内容目录

Java8 出来有一年多的时间了 ,在最开始出来的时候 ,我只是简单的看了一下他的文档，并且看了一下其中的一些例子，并没有将自己的日常开发切换到 Java8 中，最近的项目开发中，公司要求将 JDK 的平台升级到 Java8 中，因此我觉得有必要逐渐梳理一下 Java8 中的一些新的特性，Java8 其实在 Java 的发展过程中绝对是里程碑式的存在，有很多很多新颖的编程方式，可以这么说吧，Lambda 的出现改变了 Java 的编程方式，使这门语法紧凑的纯纯面向对象的语言，开始兼容函数式编程的风格，我不想评价是不是进步（因为面向函数的语言其实已经很多了，并且很流行），但是最起码 Java 平台做出来改变，我更想说是一种演进和演变。

我们来一起学习一下 Java8 的新特性 Stream ,其中可能需要你具备一些 Lambda 的知识，一些 Collections 的知识，然后我们就快速上手吧，其中为了说明并行计算，我可能会介绍一下 JDK7 中的 Fork Join 和快速排序算法等。

一、Stream 简单介绍

Stream 是 Java8 中比较闪亮的一个新特性 ,但是它绝对不等同于 IO 包中的 Stream 和解析 XML 的 Stream，JAVA 8 中的 Stream 也不是一个容器，它绝对不是用来存储数据的，他是对 JDK 中 Collections 的一个增强，他只专注于对集合对象的便利，高效的聚合操作 ,它不仅支持串行的操作功能 ,而且还借助 JDK1.7 中的 Fork-Join 机制支持了并行模式，你无需编写任何一行并行相关的代码，就能高效方便的写出高并发的程序，尤其在现在多核 CPU 的时代，最大程度的利用 CPU 的超快计算能力显得尤为重要。

二、Stream 之 Hello World

在开始介绍 Stream 的各个使用细节的时候，我们先来快速看一个入门的示例，先有一个简单的认识，为了能够体现出来 Stream 的便捷，我们同样的需求，分别使用传统的方式和 Stream 的方式各实现一次。

2.1 需求描述

在一个水果的集合中，获取苹果这一个单品并且以价格降序的方式形成一个新的集合，这个新的集合中只存放价格，也就是 List<Integer>，虽然这个例子在实际中有些扯淡，但是我们为了简单演示一下如何使用 Stream ,所以重点关注用法 ,不用理会需求的严谨与否。

2.2 代码实现

Fruit 类，代码如下：

```
package com.wangwenjun.stream;
```

```
/**
 * Created by wangwenjun on 2015/8/8.
 */
public class Fruit {

    private final String name;

    private final double price;


    public Fruit(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}
```

FruitSelector 类，代码如下

```
package com.wangwenjun.stream;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by wangwenjun on 2015/8/8.
 */
public abstract class FruitSelector {

    protected final static String CANDIDATE_FRUIT = "apple";

    private List<Fruit> getData(){
        final List<Fruit> data = new ArrayList<Fruit>(){
            {
                add(new Fruit("apple",22.1));
                add(new Fruit("apple",22.2));
                add(new Fruit("apple",22.3));
```

```
                add(new Fruit("apple",22.4));
                add(new Fruit("apple",22.5));
                add(new Fruit("apple",22.6));
                add(new Fruit("apple",22.7));
                add(new Fruit("orange",22.8));
                add(new Fruit("orange",22.9));
                add(new Fruit("orange",23.0));
                add(new Fruit("orange",23.1));
                add(new Fruit("orange",24.2));
                add(new Fruit("orange",22.3));
                add(new Fruit("banana",22.4));
                add(new Fruit("banana",22.2));
                add(new Fruit("banana",22.2));
                add(new Fruit("banana",22.2));
                add(new Fruit("banana",22.2));
            }
        };
        return data;
    }

    public List<Double> select()
    {
        List<Fruit> fruits= getData();
        return doFilter(fruits);

    }

    protected abstract List<Double> doFilter(final List<Fruit>
fruits);
}
```

IteratorFruitSelector 类代码如下，他是使用我们传统迭代的方式去做这样的工作。

```
package com.wangwenjun.stream;

import java.util.*;


public class IteratorFruitSelector extends FruitSelector {

    @Override
    protected List<Double> doFilter(List<Fruit> fruits) {

        //get the fruit name is 'apple'
        List<Fruit> appleList = new ArrayList<>();
```

```
        Iterator<Fruit> iterator = fruits.iterator();
        for (; iterator.hasNext(); ) {
            Fruit fruit = iterator.next();
            if (fruit.getName().equals(CANDIDATE_FRUIT)) {
                appleList.add(fruit);
            }
        }

        //do sort.
        Collections.sort(appleList, (o1, o2) -> {
            if (o1.getPrice() > o2.getPrice()) return 1;
            else if (o1.getPrice() == o2.getPrice()) return 0;
            else return -1;
        });


        //do filter.
        List<Double> applePriceList = new ArrayList<>();
        for (Fruit fruit : appleList) {
            applePriceList.add(fruit.getPrice());
        }
        return applePriceList;
    }
}
```

我们使用 Stream 的方式看看，代码如何去写呢？

```
package com.wangwenjun.stream;

import java.util.Comparator;
import java.util.List;

import static java.util.stream.Collectors.toList;

/**
 * Created by wangwenjun on 2015/8/8.
 */
public class StreamFruitSelector extends FruitSelector {

    @Override
    protected List<Double> doFilter(List<Fruit> fruits) {
        return              fruits.stream().filter(f             ->
f.getName().equals(CANDIDATE_FRUIT))
```

```
                .sorted(Comparator.comparing(Fruit::getPrice))
                .map(Fruit::getPrice)
                .collect(toList());
    }
}
```

代码写完了，简单对照一下，你是不是能看出来，采用 Stream 的方式比传统的方式要简洁很多很多，代码量至少少了 2/3.

2.3 简单测试

好了，我们写一下单元测试，看看是不是两者运行情况一样呢？当然我们并没有测试性能的意思，如果你想测试性能，可以把数据量改到很大，并且 Stream 采用并行的工作模式，差异还是蛮多的。

```
package com.wangwenjun.stream;

import org.junit.Test;

import java.util.List;

/**
 * Created by wangwenjun on 2015/8/8.
 */
public class FruitSelectorTest
{

    @Test
    public void testIterator()
    {
        FruitSelector selector = new IteratorFruitSelector();
        List<Double> result = selector.select();
        System.out.println(result);
    }

    @Test
    public void testStream()
    {
        FruitSelector selector = new StreamFruitSelector();
        List<Double> result = selector.select();
        System.out.println(result);
```

```
    }
}
```

运行结果不言而喻了吧

```
[22.1, 22.2, 22.3, 22.4, 22.5, 22.6, 22.7]

[22.1, 22.2, 22.3, 22.4, 22.5, 22.6, 22.7]
```

三、Fork-Join 机制介绍

Stream 之所以可以使用并行运算，之所以能够最大程度的使用 CPU 资源，是因为他的 Fork-Join 编程模型，在本章中我们介绍一下 Fork-Join 的使用，希望读者能掌握在 Java 中如何使用 Fork-Join 这一编程模型，本章内容大致如下

- **Fork-Join 简单使用**
- **快速排序算法**
- **Fork-Join 并行快速排序**

3.1 Fork-Join 简单实用

在开始说 Fork-Join 的 API 以及原理之前，我们先来看一个非常简单的例子，让读者能够快速上手，我们在本章中我们实现一个非常简单的算法，就是计算自然数的累加结果，从 start 到 end，然后将结果输出出来即可，这个简单的再也不能简单了。

我们用传统方法实现一个，大家看一下

```
    @Test
    public void testTradition()
    {
        long startTime = System.currentTimeMillis();
        int start = 0;
        int end = 1500000;
        int sum = 0;
```

```
        for(int i = start;i<end;i++)
        {
            sum+=i;
        }
        System.out.println("Tradition          Result:"+sum+",speed
time:"+(System.currentTimeMillis()-startTime));
    }
```

好了，不想多说了，我们看一下，如何使用 Fork-Join 来实现这样的需求

```
package com.wangwenjun.forkjoin;

import java.util.concurrent.RecursiveTask;

/**
 * Created by wangwenjun on 2015/8/8.
 */
public class ConcurrencyCalculator
        extends RecursiveTask<Integer> {

    private final int start;

    private final int end;

    private final static int THRESHOLD = 5;

    public ConcurrencyCalculator(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        int result = 0;
        if ((end -start) < THRESHOLD) {
            for (int x = start; x < end; x++) {
                result += x;
            }
        } else {
            int middle = (start + end) / 2;
            ConcurrencyCalculator      leftCalculator      =      new
ConcurrencyCalculator(start, middle);
            ConcurrencyCalculator      rightCalculator      =      new
ConcurrencyCalculator(middle, end);
            leftCalculator.fork();
```

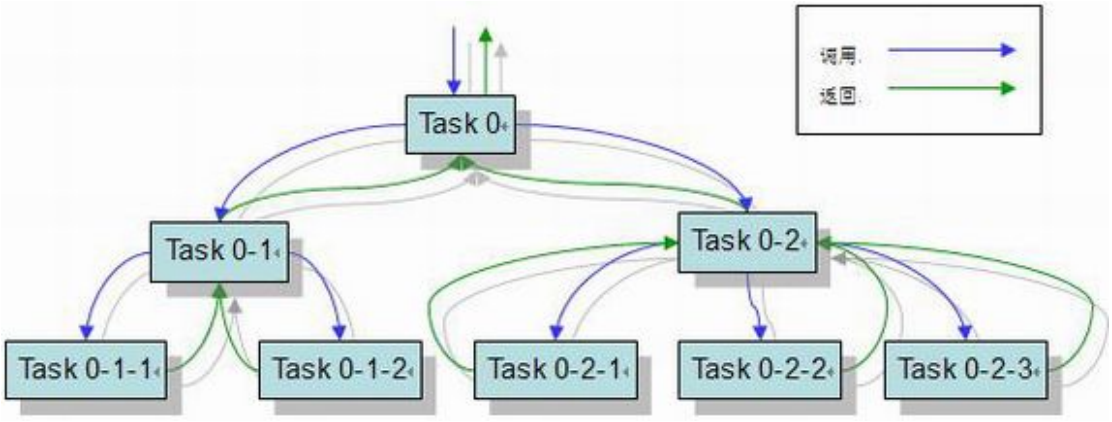
```
            rightCalculator.fork();
            result        +=        (leftCalculator.join()        +
rightCalculator.join());
        }
        return result;
    }
}
```

如果你对上面的代码理解上有些困难，别担心，慢慢的介绍了 Fork-Join 和 API 的使用之后，你就会明白如何使用了。

3.2 Fork-Join 原理以及 API 介绍

3.2.1 Fork-Join 原理介绍

我还是不太喜欢抄袭太多别人的东西来说他的基本原理，我们来举个简单的例子吧，假设你要到数据库中查询一百万条数据，数据查询结束之后，如要对每条数据进行一些额外的操作，操作有可能很简单，有可能很复杂，如果你用单线程的方式处理势必很慢，如果你用多线程的方式处理，你需要对这 100 万条数据进行拆分，那些数据交给那个线程去操作，等所有的线程操作结束之后然后在集中将结果合并，说白了就是采用分而治之的办法，Fork-Join 模型一言蔽之就是采用分而治之的方式，将一个任务分解成若干个子问题，子问题还可以继续拆分，如下图所示：

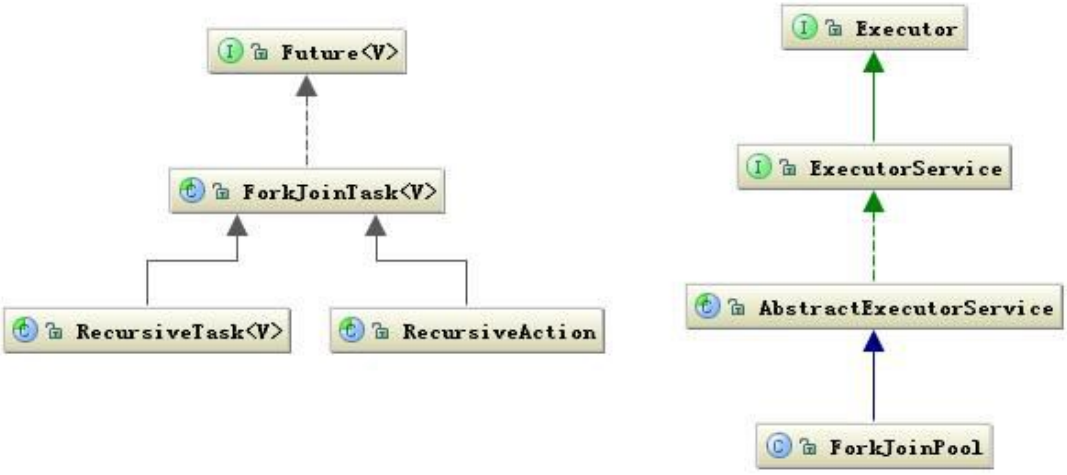


Doug Lea 大神开发的 Fork/Join 并行计算框架，集成到了 JDK 7 中，很优雅的让你远

离了自己进行加锁，释放锁，隐藏了工作线程，线程的调度等，所以你才能看到上面例子中用了很少的代码就可以模拟一个并行运算的例子。

3.2.2 Fork-Join API 的介绍

Fork-Join 主要的 API 大致如下所示



如果你计算的需要返回值就继承 RecursiveTask 抽线类，如果不需要返回值就继承 RecursiveAction 抽象类。

看到这里关于 API 的介绍和基本原理的介绍，你再回头看看上面的例子，相信就不难理解了吧，在接下来我们再来实现一个比较复杂的快速排序采用并行的方式进行，加深理解。

3.3 快速排序算法

快速排序算法是冒泡算法的改进版本，正如他的名字一样，排序速度是相当的快，快速排序由 C. A. R. Hoare 在 1962 年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

不想浪费太多的笔墨介绍快速排序算法的原理，相信很多人，尤其是 IT 行业的从业者，对于写出快速排序算法这样的基本功都是信手拈来的事情，我在这里只是将我的实现代码粘贴出来。

```
package com.wangwenjun.forkjoin;

import java.util.Random;

/**
 * Created by wangwenjun on 2015/8/9.
 */
public class QuickSort {

    private final int LEN = 10;

    public int[] prepareForData() {
        Random random = new Random(System.currentTimeMillis());
        int[] originalArray = new int[LEN];
        for (int i = 0; i < LEN; i++) {
            originalArray[i] = random.nextInt(20);
        }

        return originalArray;
    }

    public void print(String literal, int[] array) {
        System.out.println(literal);
        for (int element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    public void quickSort(int[] originalArray) {
        if (originalArray.length > 0) {
            sort(originalArray, 0, originalArray.length - 1);
        }
    }

    private void sort(int[] originalArray, int low, int high) {
        if (low < high) {
            int middle = partition(originalArray, low, high);
```

```
            sort(originalArray, low, middle - 1);
            sort(originalArray, middle + 1, high);
        }
    }

    private int partition(int[] array, int low, int high) {
        int tmp = array[low];
        while (low < high) {
            while (low < high && array[high] >= tmp) {
                high--;
            }
            array[low] = array[high];
            while (low < high && array[low] <= tmp) {
                low++;
            }
            array[high] = array[low];
        }
        array[low] = tmp;
        return low;
    }
}
```

写个测试代码，测试一下，代码如下所示：

```
package com.wangwenjun.forkjoin;

import org.junit.Test;

/**
 * Created by wangwenjun on 2015/8/9.
 */
public class QuickSortTest {

    /**
     * original literal
     */
    private final static String ORIGINAL_LITERAL = "====The original
array as below====";


    /**
     * the array after sort.
     */
    private final static String SORTED_LITERAL = "====The array after
sorted as below====";
```

```
    @Test
    public void testQuickSort() {
        QuickSort quickSort = new QuickSort();
        int[] originalArray = quickSort.prepareForData();
        quickSort.print(ORIGINAL_LITERAL, originalArray);
        quickSort.quickSort(originalArray);
        quickSort.print(SORTED_LITERAL, originalArray);
    }
}
```

我运行了一次，大致结果如下所示

```
====The original array as below====

12 18 12 18 2 1 7 9 14 13

====The array after sorted as below====

1 2 7 9 12 12 13 14 18 18
```

3.4 使用 Fork-Join 实现并行快速排序算法

我们在采用 Fork-Join 的方式来实现一下，看看如何使用并行的方式提高计算能力，提升性能。

代码如下所示：

```
package com.wangwenjun.forkjoin;

import java.util.List;
import java.util.concurrent.RecursiveAction;

/**
 * Created by wangwenjun on 2015/8/8.
 */
public class ForkJoinQuickSort<T extends Comparable> extends
RecursiveAction {

    private List<T> data;
    private int left;
    private int right;
```

```
    public ForkJoinQuickSort(List<T> data) {
        this.data = data;
        this.left = 0;
        this.right = data.size() - 1;
    }

    public ForkJoinQuickSort(List<T> data, int left, int right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }

    @Override
    protected void compute() {
        if (left < right) {
            int pivotIndex = left + ((right - left) / 2);

            pivotIndex = partition(pivotIndex);

            invokeAll(new ForkJoinQuickSort(data, left, pivotIndex -
1),
                    new ForkJoinQuickSort(data, pivotIndex + 1, right));
        }

    }

    private int partition(int pivotIndex) {
        T pivotValue = data.get(pivotIndex);

        swap(pivotIndex, right);

        int storeIndex = left;
        for (int i = left; i < right; i++) {
            if (data.get(i).compareTo(pivotValue) < 0) {
                swap(i, storeIndex);
                storeIndex++;
            }
        }

        swap(storeIndex, right);

        return storeIndex;
    }
```

```
    private void swap(int i, int j) {
        if (i != j) {
            T iValue = data.get(i);

            data.set(i, data.get(j));
            data.set(j, iValue);
        }
    }
}
```

测试代码如下所示：

```
package com.wangwenjun.forkjoin;

import org.junit.Test;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.ForkJoinPool;

/**
 * Created by wangwenjun on 2015/8/9.
 */
public class ForkJoinQuickSortTest {

    private static final int LEN = 10;

    private List<Integer> prepareForData() {
        Random random = new Random(System.currentTimeMillis());
        List<Integer> data = new ArrayList<>();
        for (int i = 0; i < LEN; i++) {
            data.add(random.nextInt(20));
        }

        return data;
    }

    private void print(String literal, List<Integer> data) {
        System.out.println(literal);
        for (int element : data) {
            System.out.print(element + " ");
        }
        System.out.println();
```

```
    }

    @Test
    public void test() {
        List<Integer> data = prepareForData();
        print("####before", data);
        ForkJoinQuickSort<Integer>       quickSort       =       new
ForkJoinQuickSort<>(data);

        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(quickSort);
        print("####after", data);
    }

}
```

运行结果如下所示：

```
####before

19 4 9 2 9 14 18 5 6 18

####after

2 4 5 6 9 9 14 18 18 19
```

3.5 Fork-Join 总结

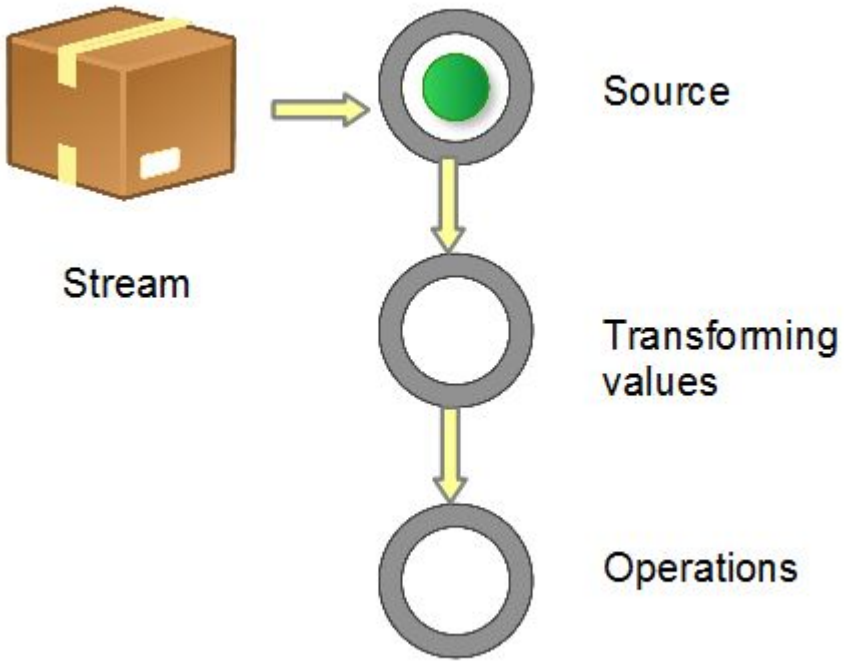
Fork-Join 编程模型出来的时间其实已经不算晚了，在 Java 1.7 版本中才被引入，做 Unix C++开发的人早都掌握该项技能了，好饭不怕晚，在我们平时的工作中他还有很多的应用场景，比如你的任务很适合进行拆分，并且比较容易进行合并，提高程序的运行速度，但是我个人建议不能将获取资源的地方使用 Fork，比如你要去网络读数据或者从数据库中读取数据，分开多个任务会导致网络以及数据库的压力，将处理过程 Fork 是一个不错的选择，获取数据除非特别需要，否则不要使用 Fork 增加并行，对资源提供者也会是一个不小的压力。

四、Stream 使用详解

好了，背景知识介绍完成，并且我们在最开始也对 Stream 有了一个大致的了解，在本章中我们详细的介绍一下 Stream，每一个示例都会有相应的代码配合，让读者理解更加的透彻。

对 Stream 的使用就是实现一个 filter-map-reduce 过程，产生一个最终结果，或者导致一个副作用（side effect），当我们使用流的时候，通常会包括三个基本的步骤：

- 获取数据
- 转换数据，每次的转换不会改变原有 Stream 而是会返回一个新的 Stream，并且转换可以有多次的转换。
- 执行操作获取想要的结果



4.1 如何获得 Stream

获取流的方式有很多种，我们在这本节中，逐个的介绍，我将代码都放在 junit 测试代码中。

4.1.1 From Collections

```
@Test
public void fromCollections()
{
    List<String> list = Collections.emptyList();
    Stream<String> stream = list.stream();
    Stream<String> parallelStream = list.parallelStream();
}
```

4.1.2 From Arrays

```
@Test
public void fromArrays()
{
    int[] array = {1,2,3,4,5};
    IntStream stream = Arrays.stream(array);
    Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5, 6);
}
```

4.1.3 From Static Factory

```
@Test
public void fromArrays()
{
    int[] array = {1,2,3,4,5};
    IntStream stream = Arrays.stream(array);
    Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5, 6);
}
```

4.1.4 From Files

```
@Test
public void fromFiles()
```

```
    {
        Path path = Paths.get("");
        try {
            Stream<Path> stream = Files.walk(path);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

4.1.5 Build By Yourself

除了一些常见的方式获取 Stream 之外，我们还可以自己构造如何获取一个 Stream，这种情形通常用于随机数、常量的 Stream，或者需要前后元素间维持着某种状态信息的 Stream。把 Supplier 实例传递给 Stream.generate() 生成的 Stream，默认是串行（相对 parallel 而言）但无序的（相对 ordered 而言）。由于它是无限的，在管道中，必须利用 limit 之类的操作限制 Stream 大小。

```
@Test
    public void generateByYourself() {
        Random random = new Random(System.currentTimeMillis());
        Supplier<Integer> supplierRandom = random::nextInt;

Stream.generate(supplierRandom).limit(10).forEach((e)->System.out
.println(e));
    }
```

好，来个更加复杂的，我们自定义一个 Supplier，来演示一下如何自己 generate 一个 Stream，为了方便代码展示，我写了两个局部内部类，都在函数内部定义的两个类，代码如下所示：

```
@Test
    public void generateByExpandSupplier() {
        class Member {
            private int id;
            private String name;

            public Member(int id, String name) {
                Member.this.id = id;
                Member.this.name = name;
```

```
            }

            public int getId() {
                return id;
            }

            public String getName() {
                return name;
            }
        }

        class MemberSupplier implements Supplier<Member> {

            private int index = 0;
            private Random random = new
Random(System.currentTimeMillis());

            @Override
            public Member get() {
                return new Member(index = random.nextInt(100),
"Member#" + index);
            }
        }

        Stream.generate(new MemberSupplier()).limit(20).forEach((m)
-> System.out.println(m.getId() + ":" + m.getName()));
    }
```

运行结果如下所示：

```
42:Member#42

30:Member#30

38:Member#38

12:Member#12

15:Member#15

……
```

4.1.6 Others

- **java.util.Spliterator**
- **Random.ints()**
- **BitSet.stream()**
- **Pattern.splitAsStream(java.lang.CharSequence)**
- **JarFile.stream()**

4.2 Stream 的操作分类

掌握了如何获取 Stream，我们本节中来看看如何对 Stream 进行操作，Stream 的操作大致分为两类

Intermediate：一个流可以后面跟随零个或多个 intermediate 操作。其目的主要是打开流，做出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用。这类操作都是惰性化的（lazy），就是说，仅仅调用到这类方法，并没有真正开始流的遍历。

Terminal：一个流只能有一个 terminal 操作，当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。Terminal 操作的执行，才会真正开始流的遍历，并且会生成一个结果，或者一个 side effect。

还有一种操作被称为 short-circuiting。用以指：

对于一个 intermediate 操作，如果它接受的是一个无限大（infinite/unbounded）的 Stream，但返回一个有限的新 Stream。

对于一个 terminal 操作，如果它接受的是一个无限大的 Stream，但能在有限的时间计算出结果。

当操作一个无限大的 Stream，而又希望在有限时间内完成操作，则在管道内拥有一个

short-circuiting 操作是必要非充分条件。

4.3 Stream 的操作实战

了解了 Stream 的分类，本节中我们一起来实战一下对其所涉及的每个方法进代码演示。

4.3.1 Intermediate 操作

Intermediate 的操作方法大致有如下几个：

- **map (mapToInt, flatMap 等)**
- **Filter**
- **Distinct**
- **Sorted**
- **Peek**
- **Limit**
- **Skip**
- **Parallel**
- **Sequential**
- **unordered**

4.3.1.1 map

```
@Test
    public void testMap() {
        int[] array = {1, 2, 3, 4, 5};
        IntStream.of(array).map(e   ->   e   *   10).forEach((e)   ->
System.out.println(e));
    }
```

上面的代码是对每一个数组的元素都增加了十倍并且打印出来。

4.3.1.2 filter

```
@Test
   public void testFilter() {
       Integer[] array = {1, 2, 3, 4, 5};
       List<Integer> result = Stream.of(array).filter(e -> e > 
3).collect(Collectors.toList());
       assertEquals(2,result.size());
   }
```

上面的代码是过滤掉比 3 小的数据，并且形成了一个新的 Integer 列表。

4.3.1.3 Distinct

```
   @Test
   public void testDistinct() {
       Integer[] array = {1, 1, 2, 3, 4, 5, 6, 5};
       assertEquals(8, array.length);
       List<Integer> result = 
Stream.of(array).distinct().collect(Collectors.toList());
       assertEquals(6, result.size());
   }
```

上面的代码过滤掉了重复的元素。

4.3.1.4 Sorted

```
  @Test
   public void testSorted() {
       Integer[] array = {3,4,6,1,8,2};

Stream.of(array).sorted().forEach(e->System.out.println(e));
   }
```

Sorted 是对流中的元素进行升序排列，并且形成一个新的流，如果你想有自己的排序规则，请实现 Compare 接口并且传给 sorted。

4.3.1.5 Peek

```
   @Test
   public void testPeek() {
       Integer[] array = {3,4,6,1,8,2};
       List<Integer> result = Stream.of(array).filter(e -> e % 2 == 
0).peek(e -> System.out.println(e)).collect(Collectors.toList());
```

```
        System.out.println(result);
    }
```

Peek 有点类似于 forEach，但是他不会中断这个 Stream，体会一下上面的代码。

4.3.1.6 Limit

```
     @Test
    public void testLimit() {
        Integer[] array = {3,4,6,1,8,2};
        Stream.of(array).limit(4).forEach(e                    ->
System.out.println(e));
    }
```

只获取流中的前四个元素，并且形成一个新的流返回。

4.3.1.7 Skip

```
    @Test
    public void testSkip() {
        Integer[] array = {3,4,6,1,8,2};
        Stream.of(array).skip(3).forEach(e                     ->
System.out.println(e));
    }
```

Limit 是保留前面的几个元素形成一个新的 ,而 skip 则是跳过前面的几个元素形成一个新的流。

4.3.1.8 Parallel

```
    @Test
    public void testParallel() {
        Integer[] array = {3,4,6,1,8,2};
        Stream.of(array).parallel().forEach(e                  ->
System.out.println(e));
    }
```

产生一个并行计算的流并且返回。

4.3.1.9 Sequential

```
    @Test
    public void testDistinct() {
        Integer[] array = {1, 1, 2, 3, 4, 5, 6, 5};
        assertEquals(8, array.length);
        List<Integer>                  result                  =
Stream.of(array).distinct().collect(Collectors.toList());
```

```
        assertEquals(6, result.size());
    }
```

和 Parallel 相反，返回一个串行执行的 Stream，有可能返回的是自己（this）。

4.3.2 Terminal

我们之前说过了 Terminal 会中断整个流的操作，Stream 的 Terminal 操作有如下几个，有些我们已经演示过了，就不给出代码占用版面了。

- **forEach**
- **forEachOrdered**
- **toArray**
- **Reduce**
- **Collect**
- **Min**
- **Max**
- **Count**
- **anyMatch**
- **allMatch**
- **noneMatch**
- **findFirst**
- **findAny**
- **iterator**

4.3.2.1 toArray

```
    @Test
    public void testForEachOrdered() {
        Integer[] array = {1, 2, 3, 4, 5, 6};
```

```
        Object[] result = Stream.of(array).filter(e -> e > 
4).toArray();
        assertEquals(2,result.length);
    }
```

返回一个新的数组。

4.3.2.2 reduce

这个方法的主要作用是把 Stream 元素组合起来。它提供一个起始值（种子），然后依照运算规则（BinaryOperator），和前面 Stream 的第一个、第二个、第 n 个元素组合。从这个意义上说，字符串拼接、数值的 sum、min、max、average 都是特殊的 reduce。例如 Stream 的 sum 就相当于

Integer sum = integers.reduce(0, (a, b) -> a+b); 或

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

也有没有起始值的情况，这时会把 Stream 的前面两个元素组合起来，返回的是 Optional。

```
   @Test
    public void testReduce() {
        String reduceResult = Stream.of("W", "a", "n", "g").reduce("", 
String::concat);
        assertEquals("Wang",reduceResult);
    }
```

4.3.2.3 max，min，count，sum

比较简单，不演示了，猜都能猜出来。

4.3.2.4 anyMatch

Stream 中的元素只要有一个满足条件则就返回 true，否则返回 false

```
   @Test
    public void testAnyMatch() {
        Integer[] array = {1, 2, 3, 4, 5};
        boolean result = Stream.of(array).anyMatch(e -> e > 3);
        assertTrue(result);
    }
```

4.3.2.5 allMatch

所有的都必须满足条件

4.3.2.6 noneMatch

所有的都不满足条件

4.3.2.7 findFirst

```
@Test
public void testFindFirst() {
    Integer[] array = {1, 2, 3, 4, 5};
    Integer result = Stream.of(array).findFirst().get();
    assertEquals(1,result.intValue());
}
```

返回流中的第一个满足条件的数据返回。

4.3.2.8 findAny

和 FindFirst 一样都是返回一个数据，但是他是返回任意的一个元素，所以之前我们的断言方式有可能会失败。

```
@Test
public void testFindAny() {
    Integer[] array = {1, 2, 3, 4, 5};
    Integer result = Stream.of(array).findAny().get();
    System.out.println(result);
}
```

4.3.2.9 iterator

```
@Test
public void testIterator() {
    Integer[] array = {1, 2, 3, 4, 5};
    Iterator<Integer> iterator = Stream.of(array).iterator();
}
```

返回我们熟悉的迭代器 Iterator，不做解释，相信你会明白吧。

五、总结

本来最近在写《Restful 实战》一书 ,但是由于工作的需要 ,暂时写一个小册子的 Stream 作为插曲，看到的朋友们如果有问题可以一起讨论，学习交流群为：208286530

这本应该我的第十二本电子书了，这本是最短的一个，其他的都在一两百页左右，相信有些朋友都读到过我写的电子书，也有很多帮我指出了不少问题，大家一起交流进步吧。

另外有一个网站，是我自己业余时间运营的，希望大家收藏起来，没事多看看，也算是多我的一个支持吧，网站功能目前比较单一，有时间就在开发增加功能，做 IT 不容易，做私活太累，赚不了几个钱而且很累，网站地址是：http://www.94jiankang.com