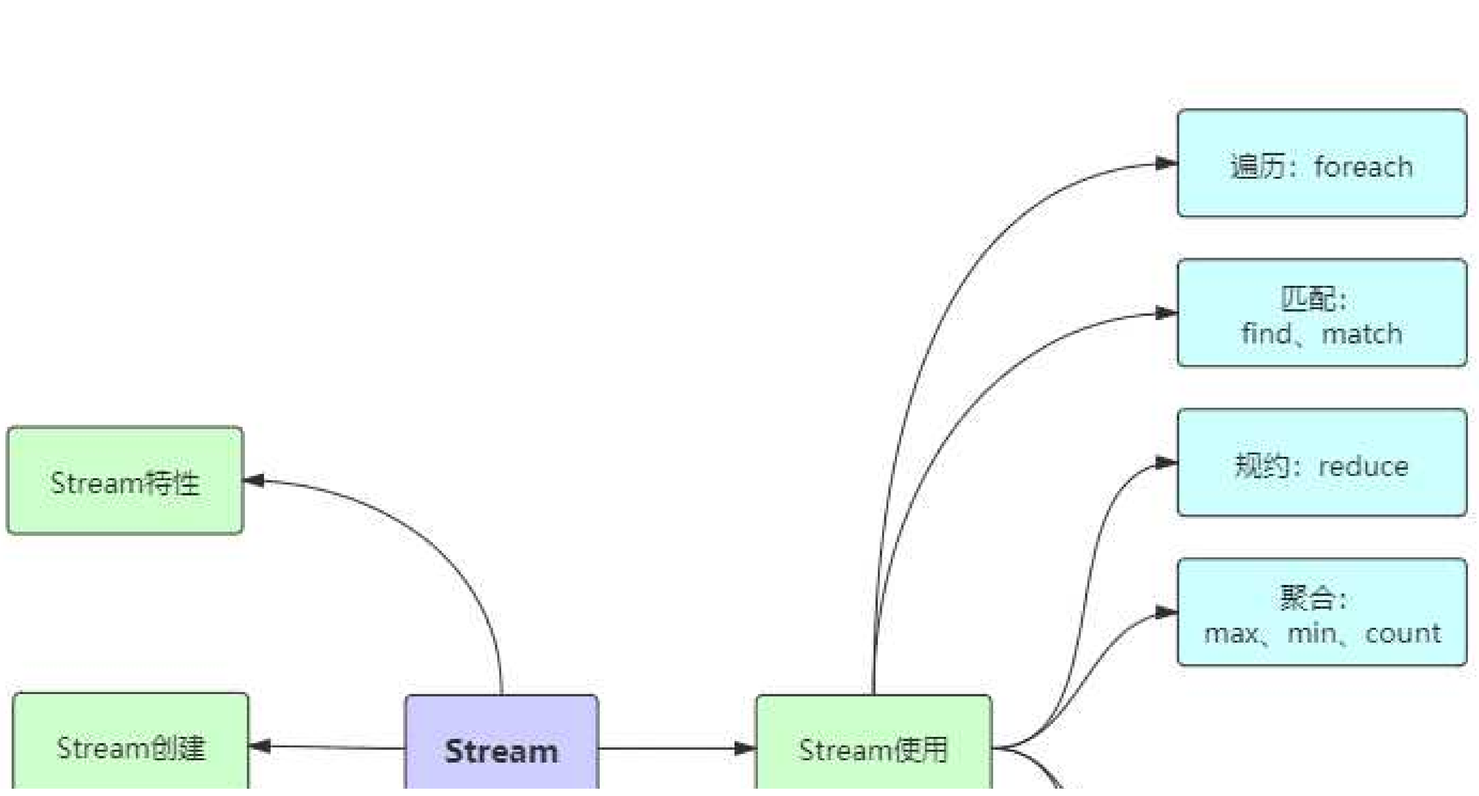
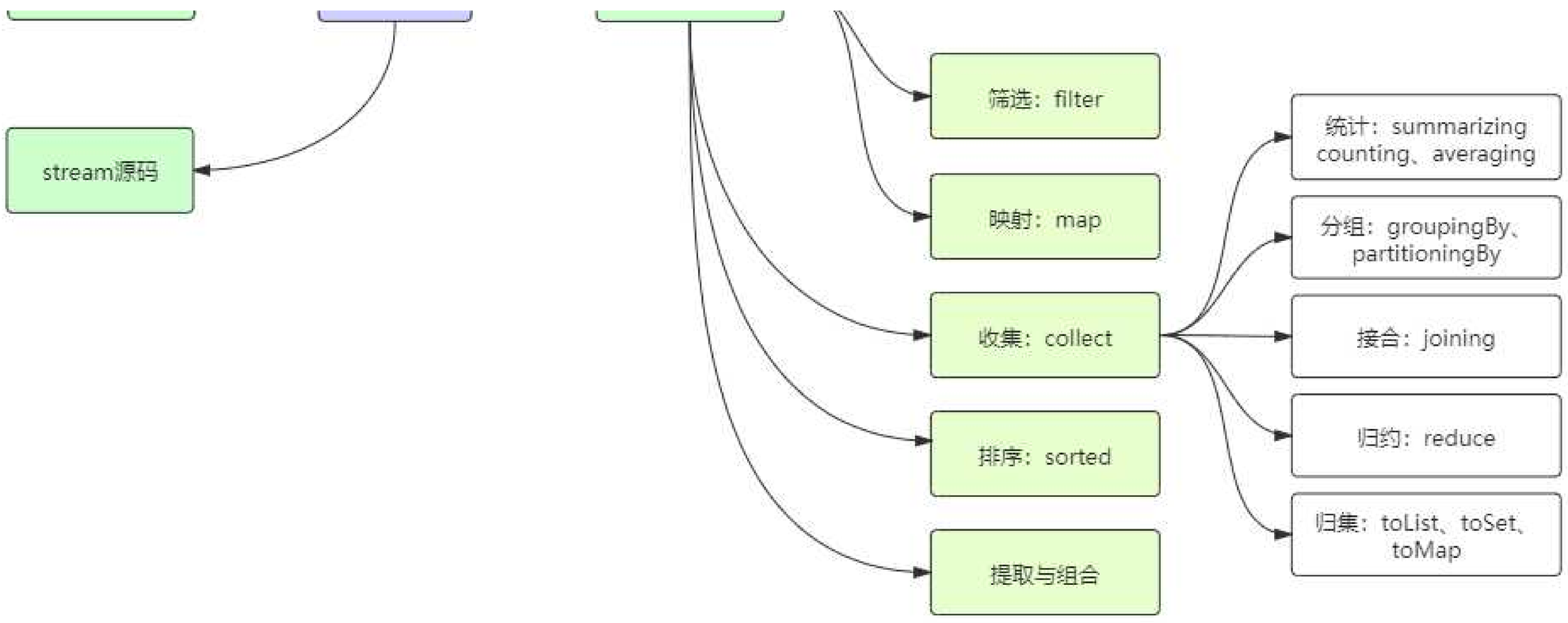
20 个实例学习 Java 8 Stream





先贴上几个案例，水平高超的同学可以挑战一下：

1. 从员工集合中筛选出salary大于8000的员工，并放置到新的集合里。

2. 统计员工的最高薪资、平均薪资、薪资之和。

3. 将员工按薪资从高到低排序，同样薪资者年龄小者在前。

4. 将员工按性别分类，将员工按性别和地区分类，将员工按薪资是否高于8000分为两部

分。

用传统的迭代处理也不是很难，但代码就显得冗余了，跟Stream相比高下立判。Java 8是一个非常成功的版本，这个版本新增的Stream，配合同版本出现的 Lambda ，给我们操作集合（Collection）提供了极大的便利。

那么什么是Stream?

Stream将要处理的元素集合看作一种流，在流的过程中，借助Stream API对流中的元素进行操作，比如：筛选、排序、聚合等。

Stream可以由数组或集合创建，对流的操作分为两种:

1. 中间操作，每次返回一个新的流，可以有

多个。

2. 终端操作，每个流只能进行一次终端操作，终端操作结束后流无法再次使用。终端操作会产生一个新的集合或值。

另外，Stream有几个特性：

1. stream不存储数据，而是按照特定的规则对数据进行计算，一般会输出结果。

2. stream不会改变数据源，通常情况下会产生一个新的集合或一个值。

3. stream具有延迟执行特性，只有调用终端操作时，中间操作才会执行。

Stream可以通过集合数组创建。

1、通过 java.util.Collection.stream() 方法

用集合创建流

```
List<String> list = Arrays.asList("a", "
// 创建一个顺序流
Stream<String> stream = list.stream();
// 创建一个并行流
Stream<String> parallelStream = list.par
```

2、使用java.util.Arrays.stream(T[] array)方法用数组创建流

```
int[] array={1,3,5,6,8};
IntStream stream = Arrays.stream(array);
```

3、使用Stream的静态方法：of()、iterate()、generate()

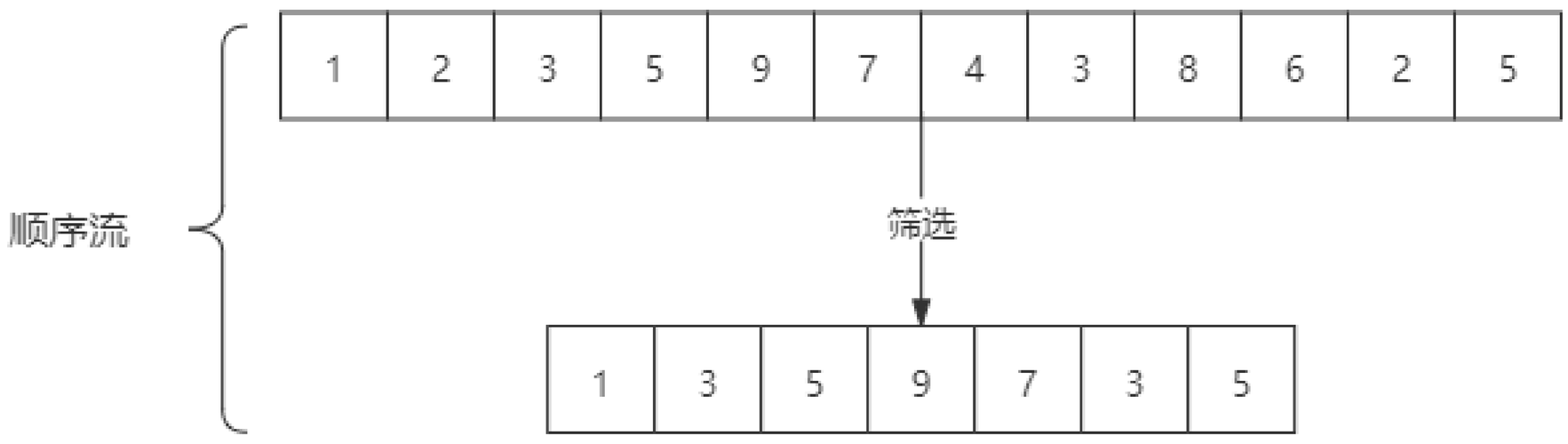
```
Stream<Integer> stream = Stream.of(1, 2,

Stream<Integer> stream2 = Stream.iterate
```

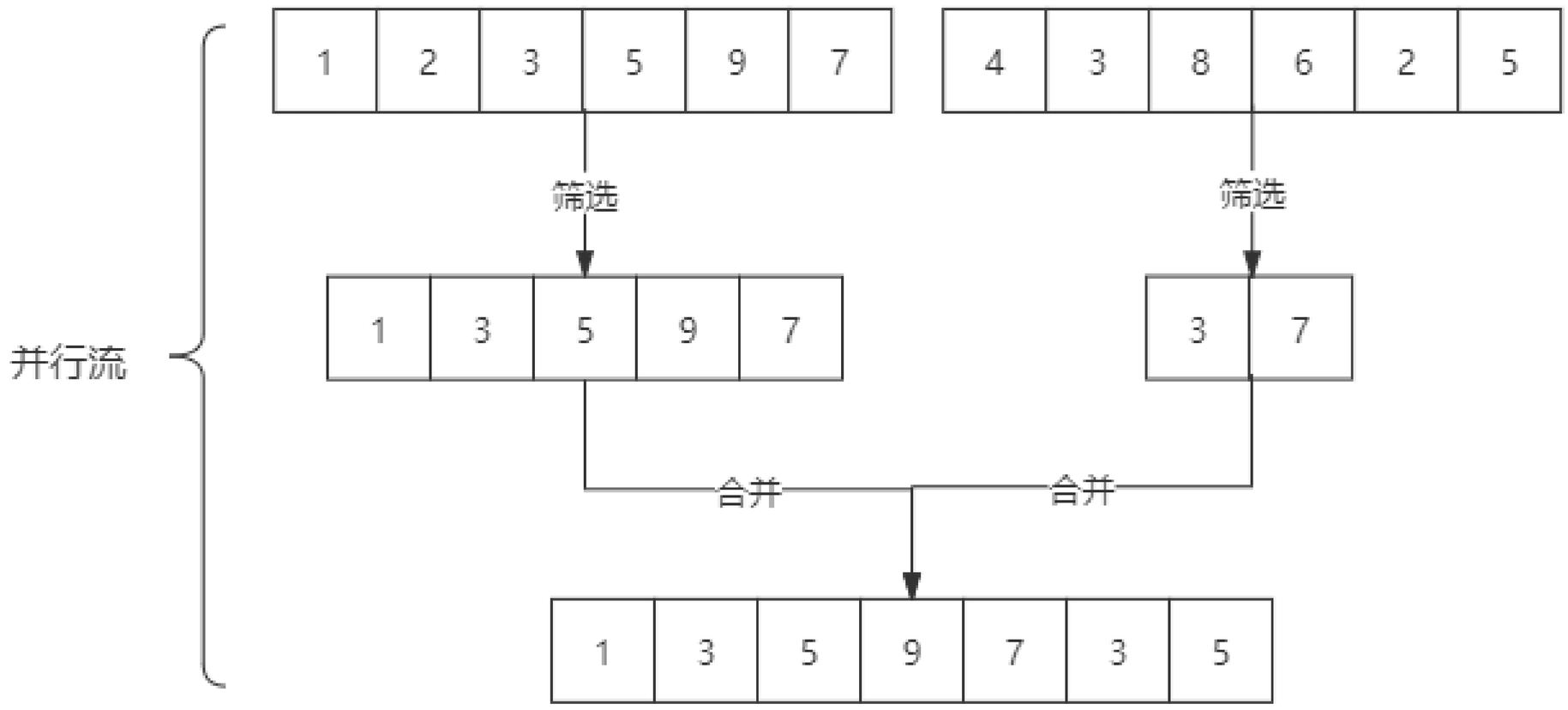
```
stream2.forEach(System.out::println);


Stream<Double> stream3 = Stream.generate
stream3.forEach(System.out::println);
```

输出结果:

> 0 3 6 9
>
> 0.6796156909271994
>
> 0.1914314208854283
>
> 0.8116932592396652

stream 和 parallelStream 的简单区分： stream是顺序流，由主线程按顺序对流执行操作，而parallelStream是并行流，内部以多线程并行执行的方式对流进行操作，但前提是流中的数据处理没有顺序要求。例如筛选集合中的奇数，两者的处理不同之处:





如果流中的数据量足够大，并行流可以加快处速度。除了直接创建并行流，还可以通过parallel()把顺序流转换成并行流：

```
Optional<Integer> findFirst = list.strea
```

在使用stream之前，先理解一个概念：

Optional 。

> Optional类是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。
>
> 更详细说明请见：菜鸟教程Java 8 Optional类

接下来，大批代码向你袭来！我将用20个案例将Stream的使用整得明明白白，只要跟着敲一遍代码，就能很好地掌握。

案例使用的员工类

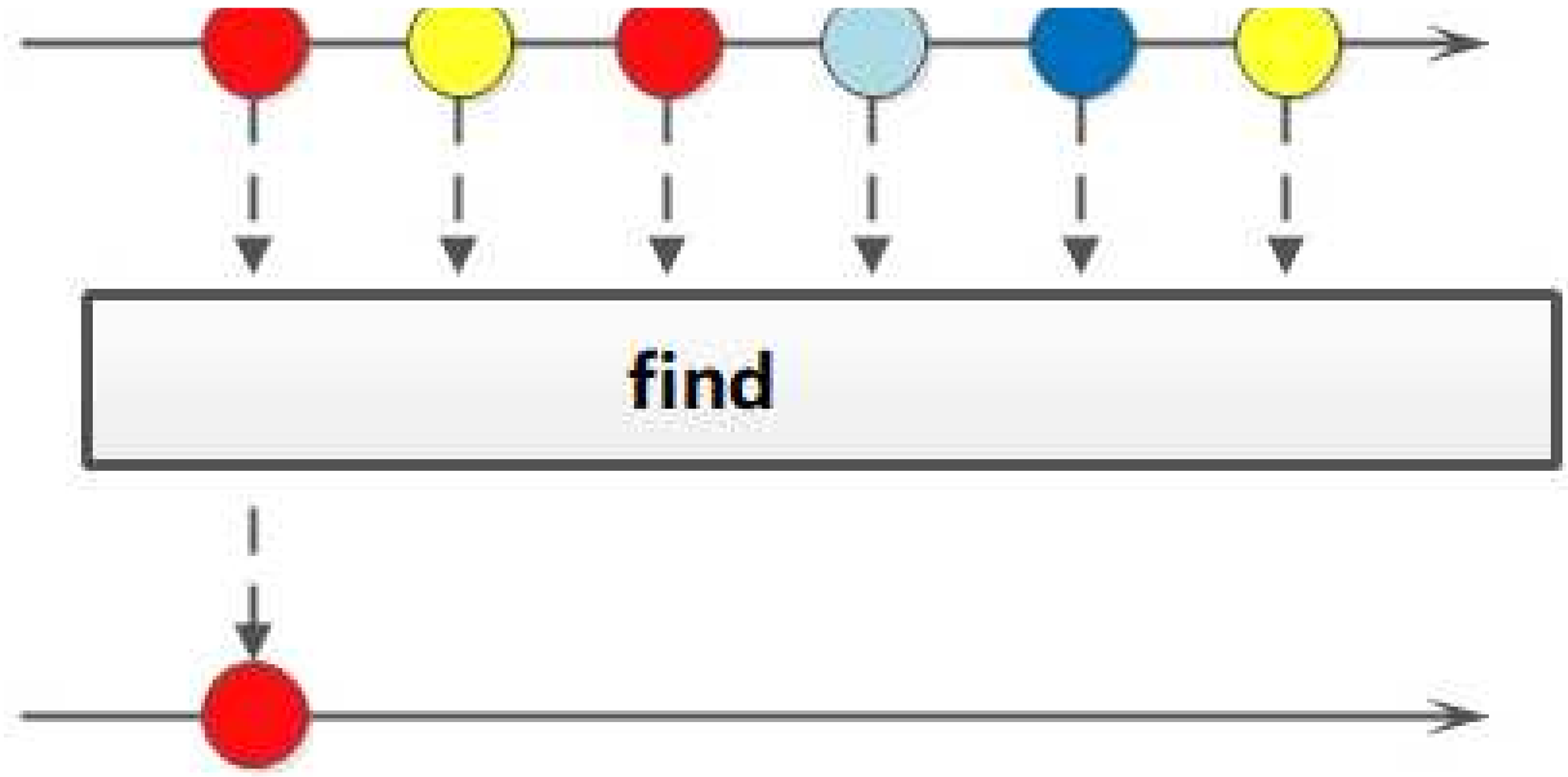
这是后面案例中使用的员工类:

```
List<Person> personList = new ArrayList<
personList.add(new Person("Tom", 8900, "
personList.add(new Person("Jack", 7000,
personList.add(new Person("Lily", 7800,
personList.add(new Person("Anni", 8200,
personList.add(new Person("Owen", 9500,
personList.add(new Person("Alisa", 7900,


class Person {
    private String name; // 姓名
    private int salary; // 薪资
    private int age; // 年龄
    private String sex; //性别
    private String area; // 地区
```

```
    // 构造方法
    public Person(String name, int salary,
        this.name = name;
        this.salary = salary;
        this.age = age;
        this.sex = sex;
        this.area = area;
    }
    // 省略了get和set，请自行添加


}
```

3.1 遍历/匹配(foreach/find/match)

Stream也是支持类似集合的遍历和匹配元素的，只是Stream中的元素是以Optional类型存在的。Stream的遍历、匹配非常简单。

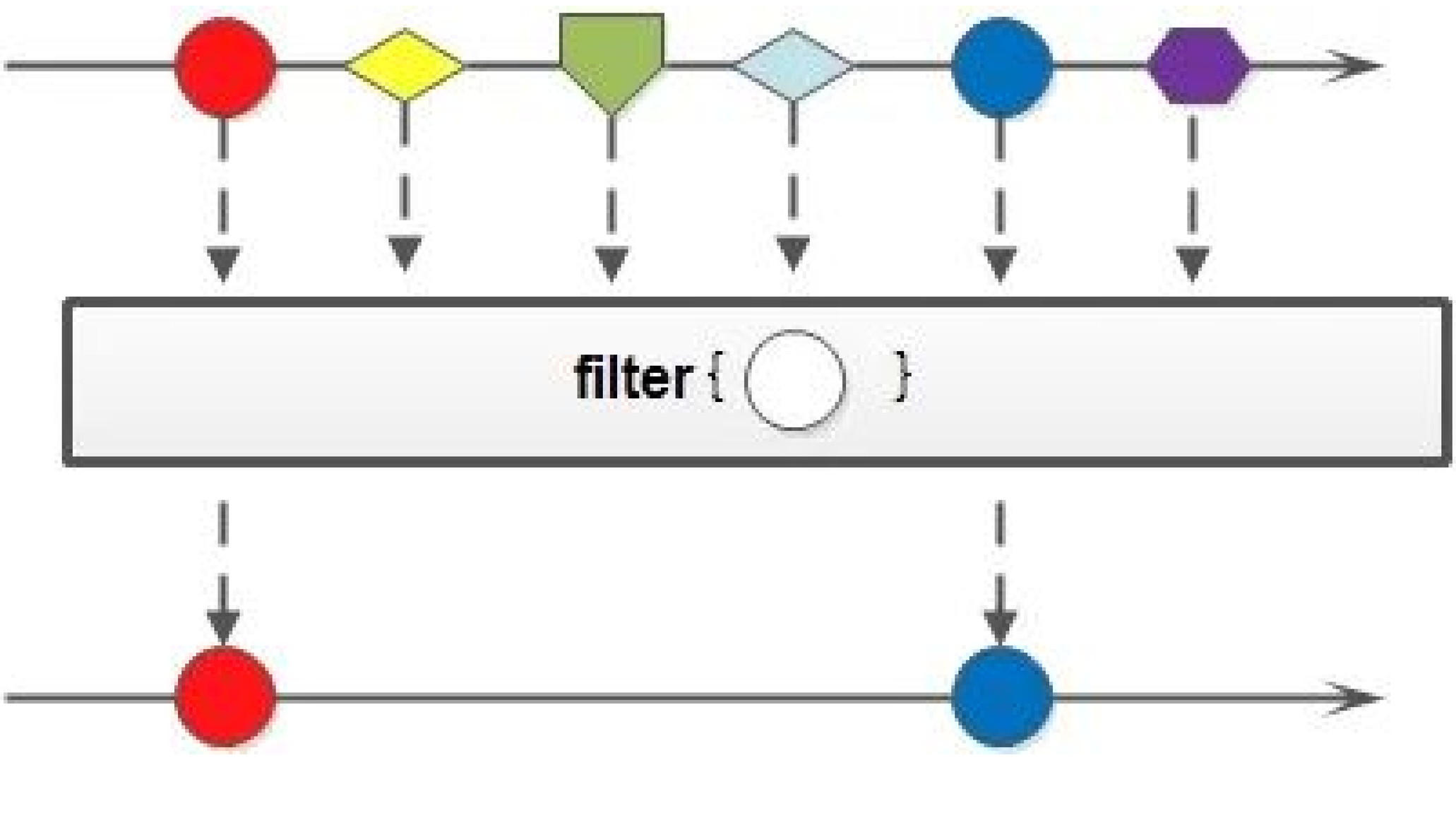



```
// import已省略，请自行添加，后面代码亦是

public class StreamTest {
    public static void main(String[] args)
        List<Integer> list = Arrays.asLi

        // 遍历输出符合条件的元素
        list.stream().filter(x -> x > 6)
        // 匹配第一个
        Optional<Integer> findFirst = li
        // 匹配任意（适用于并行流）
        Optional<Integer> findAny = list
        // 是否包含符合特定条件的元素
        boolean anyMatch = list.stream()
```

```
            System.out.println("匹配第一个值: "
            System.out.println("匹配任意一个值:
            System.out.println("是否存在大于6的
        }
}
```

3.2 筛选（filter）

筛选，是按照一定的规则校验流中的元素，将符合条件的元素提取到新的流中的操作。



案例一：筛选出Integer集合中大于7的元素，并打印出来

```
public class StreamTest {
    public static void main(String[] args)
        List<Integer> list = Arrays.asList(6
        Stream<Integer> stream = list.stream
        stream.filter(x -> x > 7).forEach(Sy
    }
}
```

预期结果：

> 89

案例二：筛选员工中工资高于8000的人，并形成新的集合。 形成新集合依赖collect（收集），后文有详细介绍。

```
public class StreamTest {
    public static void main(String[] args)
        List<Person> personList = new ArrayL
        personList.add(new Person("Tom", 890
```

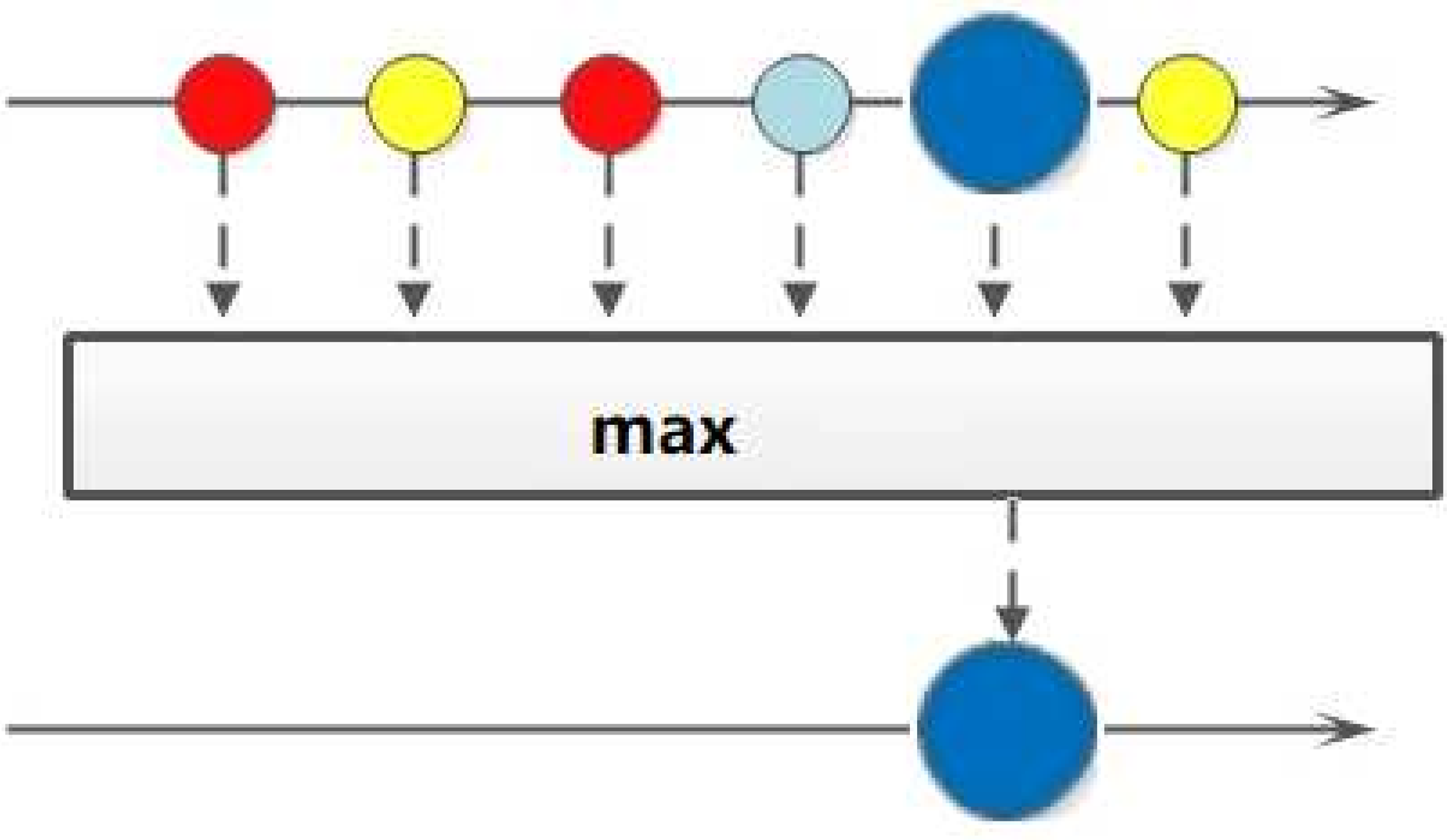
```
        personList.add(new Person("Jack", 70
        personList.add(new Person("Lily", 78
        personList.add(new Person("Anni", 82
        personList.add(new Person("Owen", 95
        personList.add(new Person("Alisa", 7

        List<String> fiterList = personList.
                .collect(Collectors.toList());
        System.out.print("高于8000的员工姓名: "
    }
}
```

运行结果:

> 高于8000的员工姓名: [Tom, Anni, Owen]

3.3 聚合（max/min/count）

max、min、count这些字眼你一定不陌生，没错，在mysql中我们常用它们进行数据统计。Java stream中也引入了这些概念和用法，极大地方便了我们对集合、数组的数据统计工作。



案例一：获取String集合中最长的元素。

```
public class StreamTest {
    public static void main(String[] args)
        List<String> list = Arrays.asList("a
```

```
        Optional<String> max = list.stream()
        System.out.println("最长的字符串: " + m
    }
}
```

输出结果:

> 最长的字符串：weoujgsd

案例二：获取Integer集合中的最大值。

```
public class StreamTest {
    public static void main(String[] args)
        List<Integer> list = Arrays.asList(7

        // 自然排序
        Optional<Integer> max = list.stream(
        // 自定义排序
        Optional<Integer> max2 = list.stream
            @Override
```

```
        public int compare(Integer o1, Int
            return o1.compareTo(o2);
        }
    });
    System.out.println("自然排序的最大值: "
    System.out.println("自定义排序的最大值:
    }
}
```

输出结果:

> 自然排序的最大值：11
>
> 自定义排序的最大值：11

案例三：获取员工工资最高的人。

```
public class StreamTest {
    public static void main(String[] args)
        List<Person> personList = new ArrayL
        personList.add(new Person("Tom", 890
```

```
        personList.add(new Person("Jack", 70
        personList.add(new Person("Lily", 78
        personList.add(new Person("Anni", 82
        personList.add(new Person("Owen", 95
        personList.add(new Person("Alisa", 7

        Optional<Person> max = personList.st
        System.out.println("员工工资最大值: " +
    }
}
```

输出结果:

> 员工工资最大值：9500

案例四：计算Integer集合中大于6的元素的个数。

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

```
public class StreamTest {
    public static void main(String[] args)
        List<Integer> list = Arrays.asList(7


        long count = list.stream().filter(x
        System.out.println("list中大于6的元素个
    }
}
```

输出结果:

> list中大于6的元素个数：4

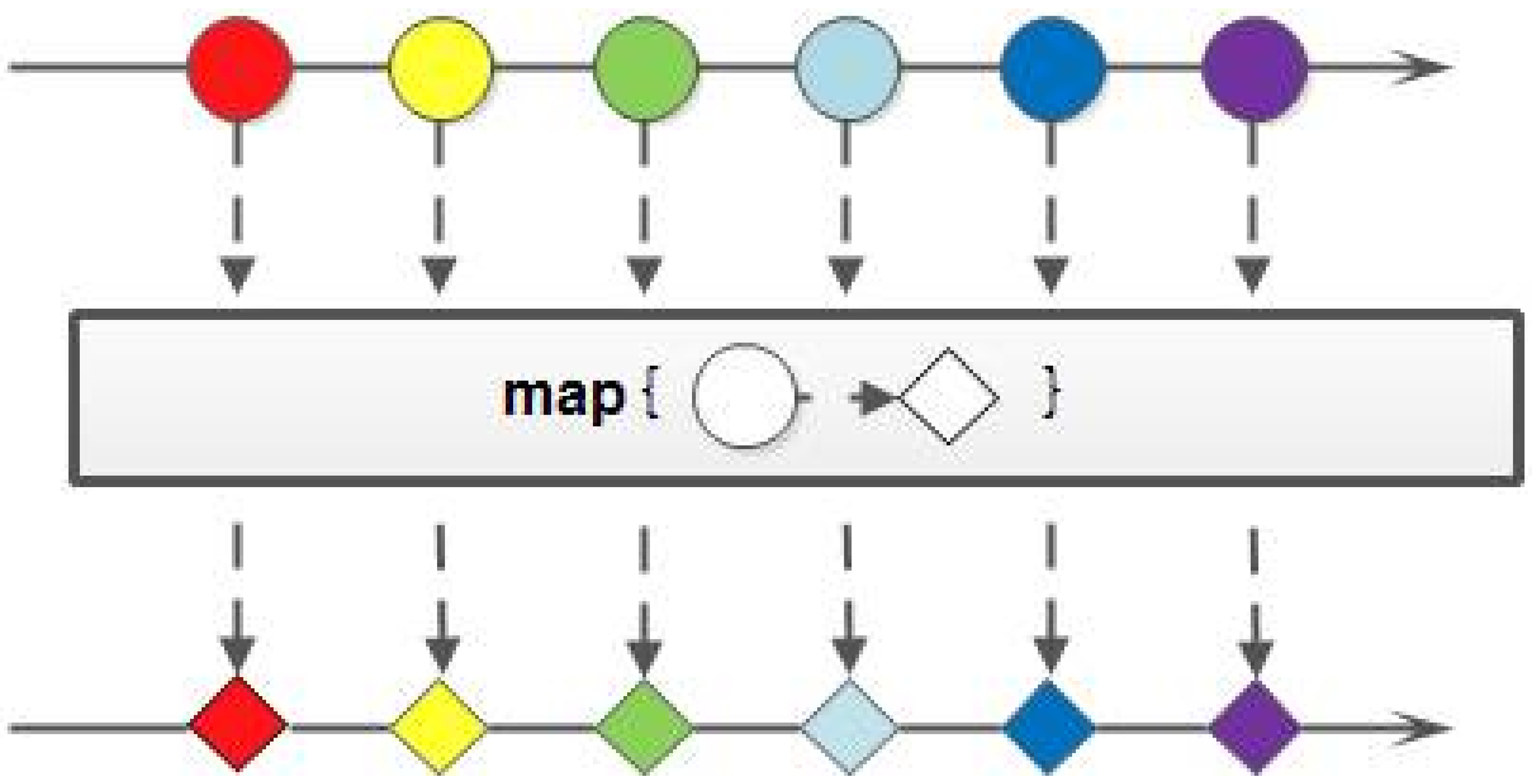
3.4 映射(map/flatMap)

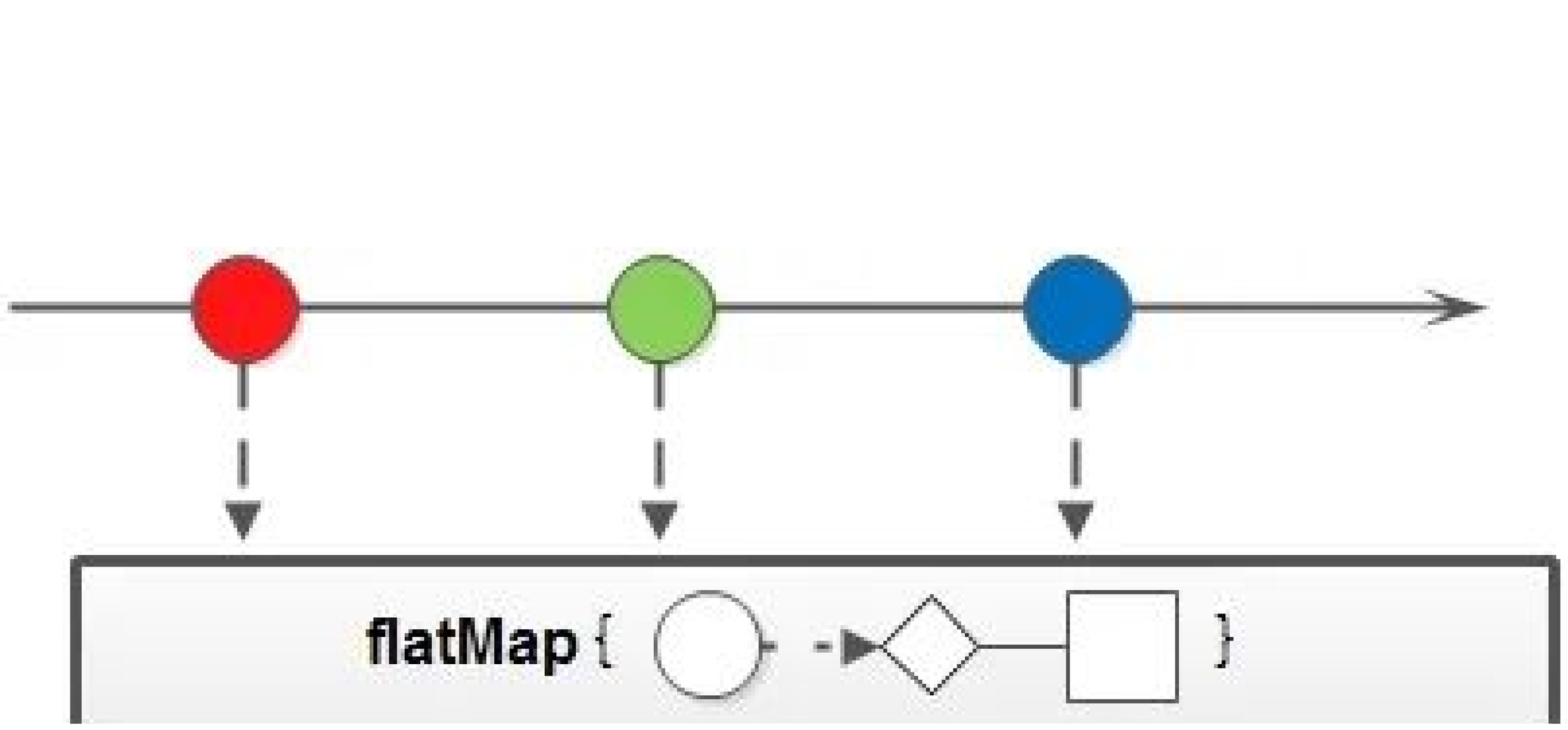
映射，可以将一个流的元素按照一定的映射规则映射到另一个流中。分为map和flatMap:

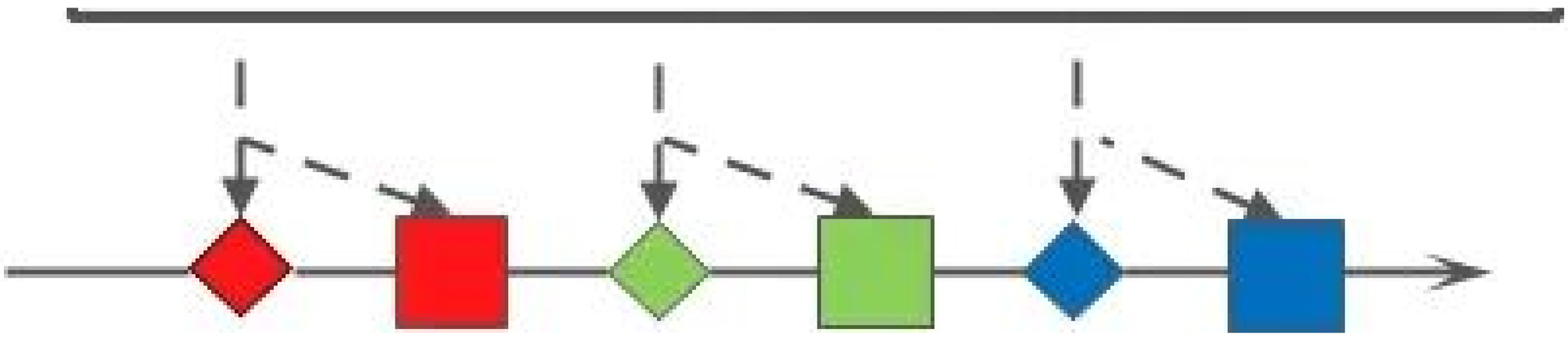
- map：接收一个函数作为参数，该函数会

被应用到每个元素上，并将其映射成一个新的元素。

- flatMap：接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。







案例一：英文字符串数组的元素全部改为大写。整数数组每个元素+3。

```
public class StreamTest {
    public static void main(String[] args)
        String[] strArr = { "abcd", "bcdd",
        List<String> strList = Arrays.stream


        List<Integer> intList = Arrays.asLis
        List<Integer> intListNew = intList.s


        System.out.println("每个元素大写: " + s
        System.out.println("每个元素+3: " + ir
    }
}
```

输出结果:

每个元素大写：[ABCD, BCDD, DEFDE, FTR]

每个元素+3：[4, 6, 8, 10, 12, 14]

案例二：将员工的薪资全部增加1000。

```
public class StreamTest {
    public static void main(String[] a
        List<Person> personList = new Ar
        personList.add(new Person("Tom",
        personList.add(new Person("Jack"
        personList.add(new Person("Lily"
        personList.add(new Person("Anni"
        personList.add(new Person("Owen"
        personList.add(new Person("Alisa

        // 不改变原来员工集合的方式
        List<Person> personListNew = per
            Person personNew = new Person(
```

```
            personNew.setSalary(person.get
            return personNew;
        }).collect(Collectors.toList());
        System.out.println("一次改动前: " +
        System.out.println("一次改动后: " +

        // 改变原来员工集合的方式
        List<Person> personListNew2 = pe
            person.setSalary(person.getSal
            return person;
        }).collect(Collectors.toList());
        System.out.println("二次改动前: " +
            System.out.println("二次改动后: " +
    }
}
```

输出结果:

> 一次改动前: Tom->8900
>
> 一次改动后: Tom->18900

> 二次改动前：Tom->18900
>
> 二次改动后：Tom->18900

案例三：将两个字符数组合并成一个新的字符数组。

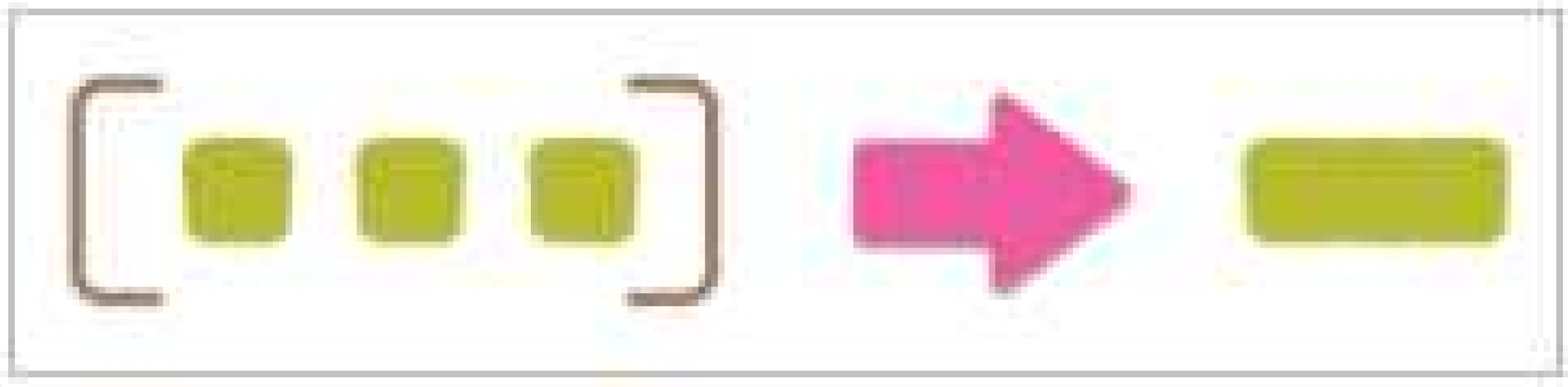
```
public class StreamTest {
    public static void main(String[] args)
        List<String> list = Arrays.asList("m
        List<String> listNew = list.stream()
            // 将每个元素转换成一个stream
            String[] split = s.split(",");
            Stream<String> s2 = Arrays.stream(
            return s2;
        }).collect(Collectors.toList());

        System.out.println("处理前的集合: "  + l
        System.out.println("处理后的集合: "  + l
    }
}
```

输出结果：

> 处理前的集合：[m-k-l-a, 1-3-5]
>
> 处理后的集合：[m, k, l, a, 1, 3, 5]

3.5 归约(reduce)

归约，也称缩减，顾名思义，是把一个流缩减成一个值，能实现对集合求和、求乘积和求最值操作。



案例一：求Integer集合的元素之和、乘积和最大值。

```
public class StreamTest {
    public static void main(String[] args)
```

```
        List<Integer> list = Arrays.asList(1
        // 求和方式1
        Optional<Integer> sum = list.stream(
        // 求和方式2
        Optional<Integer> sum2 = list.stream
        // 求和方式3
        Integer sum3 = list.stream().reduce(

        // 求乘积
        Optional<Integer> product = list.str

        // 求最大值方式1
        Optional<Integer> max = list.stream(
        // 求最大值写法2
        Integer max2 = list.stream().reduce(

        System.out.println("list求和: " + sum
        System.out.println("list求积: " + pro
        System.out.println("list求和: " + max
    }
}
```

输出结果：

> list求和：29,29,29
>
> list求积：2112
>
> list求和：11,11

案例二：求所有员工的工资之和和最高工资。

```
public class StreamTest {
    public static void main(String[] args)
        List<Person> personList = new ArrayL
        personList.add(new Person("Tom", 890
        personList.add(new Person("Jack", 70
        personList.add(new Person("Lily", 78
        personList.add(new Person("Anni", 82
        personList.add(new Person("Owen", 95
        personList.add(new Person("Alisa", 7

        // 求工资之和方式1：
```

```
        Optional<Integer> sumSalary = person
        // 求工资之和方式2:
        Integer sumSalary2 = personList.stre
                (sum1, sum2) -> sum1 + sum2);
        // 求工资之和方式3:
        Integer sumSalary3 = personList.stre


        // 求最高工资方式1:
        Integer maxSalary = personList.strea
                Integer::max);
        // 求最高工资方式2:
        Integer maxSalary2 = personList.stre
                (max1, max2) -> max1 > max2 ? ma


        System.out.println("工资之和: " + sumS
        System.out.println("最高工资: " + maxS
    }
}
```

输出结果:

> 工资之和：49300,49300,49300
>
> 最高工资：9500,9500

3.6 收集(collect)

collect，收集，可以说是内容最繁多、功能最丰富的部分了。从字面上去理解，就是把一个流收集起来，最终可以是收集成一个值也可以收集成一个新的集合。

> collect 主要依赖 java.util.stream.Collectors类内置的静态方法。

3.6.1 归集(toList/toSet/toMap)

因为流不存储数据，那么在流中的数据完成处理后，需要将流中的数据重新归集到新的集合里。toList、toSet和toMap比较常用，另外还有toCollection、toConcurrentMap

等复杂一些的用法。

下面用一个案例演示 toList 、 toSet 和 toMap:

```
public class StreamTest {
    public static void main(String[] args)
        List<Integer> list = Arrays.asList(1
        List<Integer> listNew = list.stream(
        Set<Integer> set = list.stream().fil

        List<Person> personList = new ArrayL
        personList.add(new Person("Tom", 890
        personList.add(new Person("Jack", 70
        personList.add(new Person("Lily", 78
        personList.add(new Person("Anni", 82

        Map<?, Person> map = personList.stre
                .collect(Collectors.toMap(Person
        System.out.println("toList:" + listN
        System.out.println("toSet:" + set);
        System.out.println("toMap:" + map);
```

```
    }
}
```

运行结果:

```
toList:  [6, 4, 6, 6, 20]
toSet:  [4, 20, 6]
toMap                                                    :
{Tom=mutest.Person@5fd0d5ae,
Anni=mutest.Person@2d98a335}
```

3.6.2 统计(count/averaging)

Collectors提供了一系列用于数据统计的静态方法:

- 计数: count
- 平　　均　　值　　:　　averagingInt　　、

averagingLong、averagingDouble

- 最值：maxBy、minBy
- 求和：summingInt、summingLong、summingDouble
- 统计以上所有：summarizingInt、summarizingLong、summarizingDouble

案例：统计员工人数、平均工资、工资总额、最高工资。

```
public class StreamTest {
    public static void main(String[] args)
        List<Person> personList = new ArrayL
        personList.add(new Person("Tom", 890
        personList.add(new Person("Jack", 70
        personList.add(new Person("Lily", 78

        // 求总数
        Long count = personList.stream().col
```

```
        // 求平均工资
        Double average = personList.stream()
        // 求最高工资
        Optional<Integer> max = personList.s
        // 求工资之和
        Integer sum = personList.stream().co
        // 一次性统计所有信息
        DoubleSummaryStatistics collect = pe


        System.out.println("员工总数: " + cour
        System.out.println("员工平均工资: " + a
        System.out.println("员工工资总和: " + s
        System.out.println("员工工资所有统计: "
    }
}
```

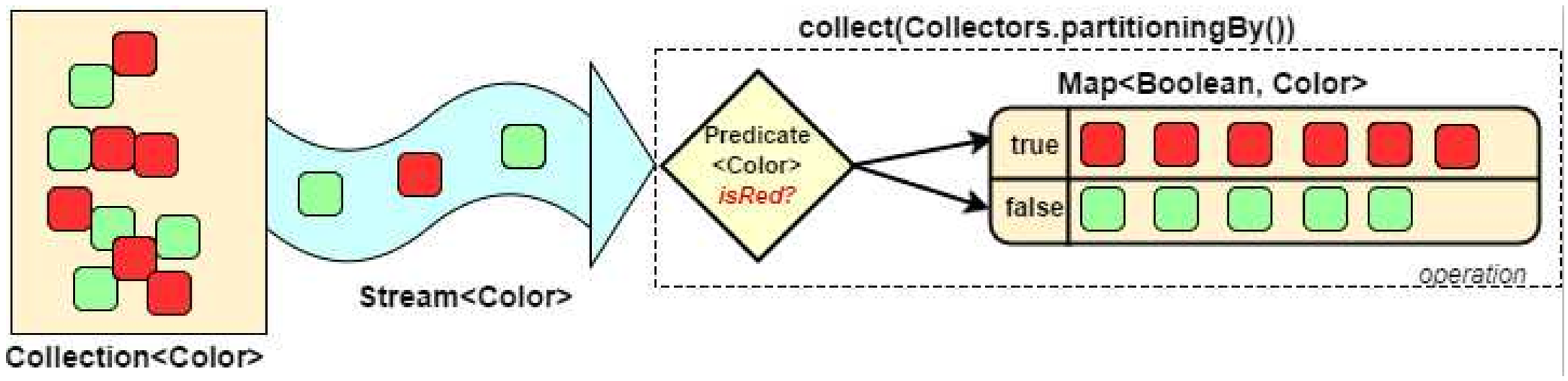
运行结果:

> 员工总数：3
>
> 员工平均工资：7900.0

> 员工工资总和：23700
>
> 员 工 工 资 所 有 统 计 ：
>
> DoubleSummaryStatistics{count=3,
>
> sum=23700.000000,min=7000.00000
>
> 0, average=7900.000000,
>
> max=8900.000000}

3.6.3 分组(partitioningBy/groupingBy)

- 分区：将stream按条件分为两个Map，比如员工按薪资是否高于8000分为两部分。
- 分组：将集合分为多个Map，比如员工按性别分组。有单级分组和多级分组。



案例：将员工按薪资是否高于8000分为两部分；将员工按性别和地区分组

```
public class StreamTest {
    public static void main(String[] a
        List<Person> personList = new Ar
        personList.add(new Person("Tom",
        personList.add(new Person("Jack"
        personList.add(new Person("Lily"
        personList.add(new Person("Anni"
        personList.add(new Person("Owen"
        personList.add(new Person("Alisa

        // 将员工按薪资是否高于8000分组
            Map<Boolean, List<Person>> p
            // 将员工按性别分组
            Map<String, List<Person>> gr
            // 将员工先按性别分组，再按地区分组
            Map<String, Map<String, List
            System.out.println("员工按薪资
```

```
            System.out.println("员工按性别
            System.out.println("员工按性别
    }
}
```

输出结果:

```
员工按薪资是否大于8000分组情况: {false=[mutes
员工按性别分组情况: {female=[mutest.Person@1
员工按性别、地区: {female={New York=[mutest
```

3.6.4 接合(joining)

joining可以将stream中的元素用特定的连接符（没有的话，则直接连接）连接成一个字符串。

```
public class StreamTest {
    public static void main(String[] args)
```

```
        List<Person> personList = new ArrayL
        personList.add(new Person("Tom", 890
        personList.add(new Person("Jack", 70
        personList.add(new Person("Lily", 78

        String names = personList.stream().m
        System.out.println("所有员工的姓名: " +
        List<String> list = Arrays.asList("A
        String string = list.stream().collec
        System.out.println("拼接后的字符串: " +
    }
}
```

运行结果:

> 所有员工的姓名: Tom,Jack,Lily
> 拼接后的字符串: A-B-C

3.6.5 归约(reducing)

Collectors类提供的reducing方法，相比于stream本身的reduce方法，增加了对自定义归约的支持。

```
public class StreamTest {
    public static void main(String[] args)
        List<Person> personList = new ArrayL
        personList.add(new Person("Tom", 890
        personList.add(new Person("Jack", 70
        personList.add(new Person("Lily", 78


        // 每个员工减去起征点后的薪资之和（这个例子:
        Integer sum = personList.stream().co
        System.out.println("员工扣税薪资总和: "


        // stream的reduce
        Optional<Integer> sum2 = personList.
        System.out.println("员工薪资总和: " + s
    }
}
```

运行结果:

> 员工扣税薪资总和：8700
> 员工薪资总和：23700

3.7 排序(sorted)

sorted，中间操作。有两种排序:

- sorted()：自然排序，流中元素需实现Comparable接口
- sorted(Comparator com)：Comparator排序器自定义排序

案例：将员工按工资由高到低（工资一样则按年龄由大到小）排序

```
public class StreamTest {
    public static void main(String[] args)
        List<Person> personList = new ArrayL

        personList.add(new Person("Sherry",
        personList.add(new Person("Tom", 890
        personList.add(new Person("Jack", 90
        personList.add(new Person("Lily", 88
        personList.add(new Person("Alisa", 9

        // 按工资升序排序（自然排序）
        List<String> newList = personList.st
                .collect(Collectors.toList());
        // 按工资倒序排序
        List<String> newList2 = personList.s
                .map(Person::getName).collect(Co
        // 先按工资再按年龄升序排序
        List<String> newList3 = personList.s
                .sorted(Comparator.comparing(Per
                .collect(Collectors.toList());
        // 先按工资再按年龄自定义排序（降序）
        List<String> newList4 = personList.s
          if (p1.getSalary() == p2.getSalary
```

```
            return p2.getAge() - p1.getAge()
        } else {
            return p2.getSalary() - p1.getSa
        }
    }).map(Person::getName).collect(Coll

    System.out.println("按工资升序排序: " +
    System.out.println("按工资降序排序: " +
    System.out.println("先按工资再按年龄升序
    System.out.println("先按工资再按年龄自定
}
```

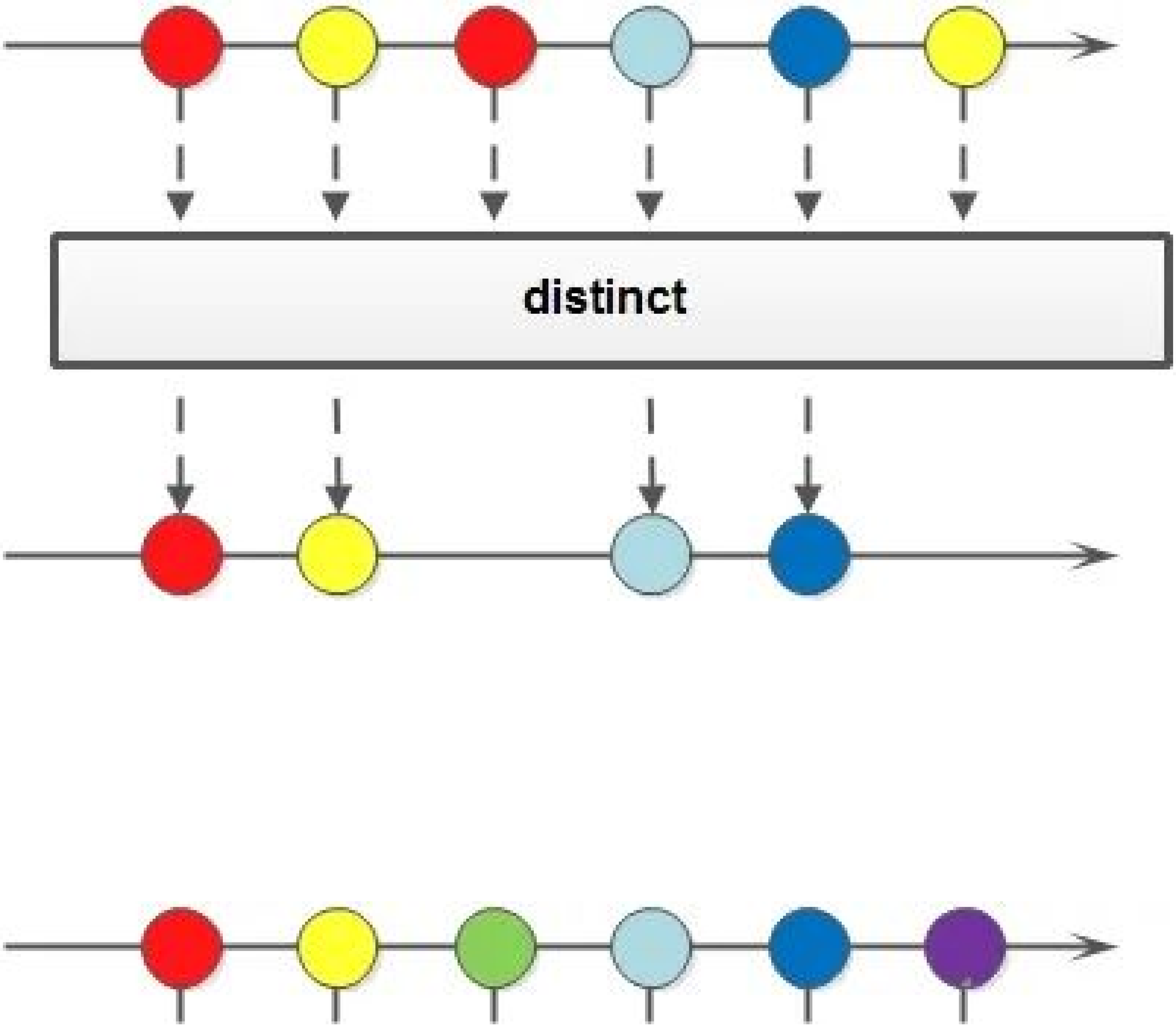
运行结果:

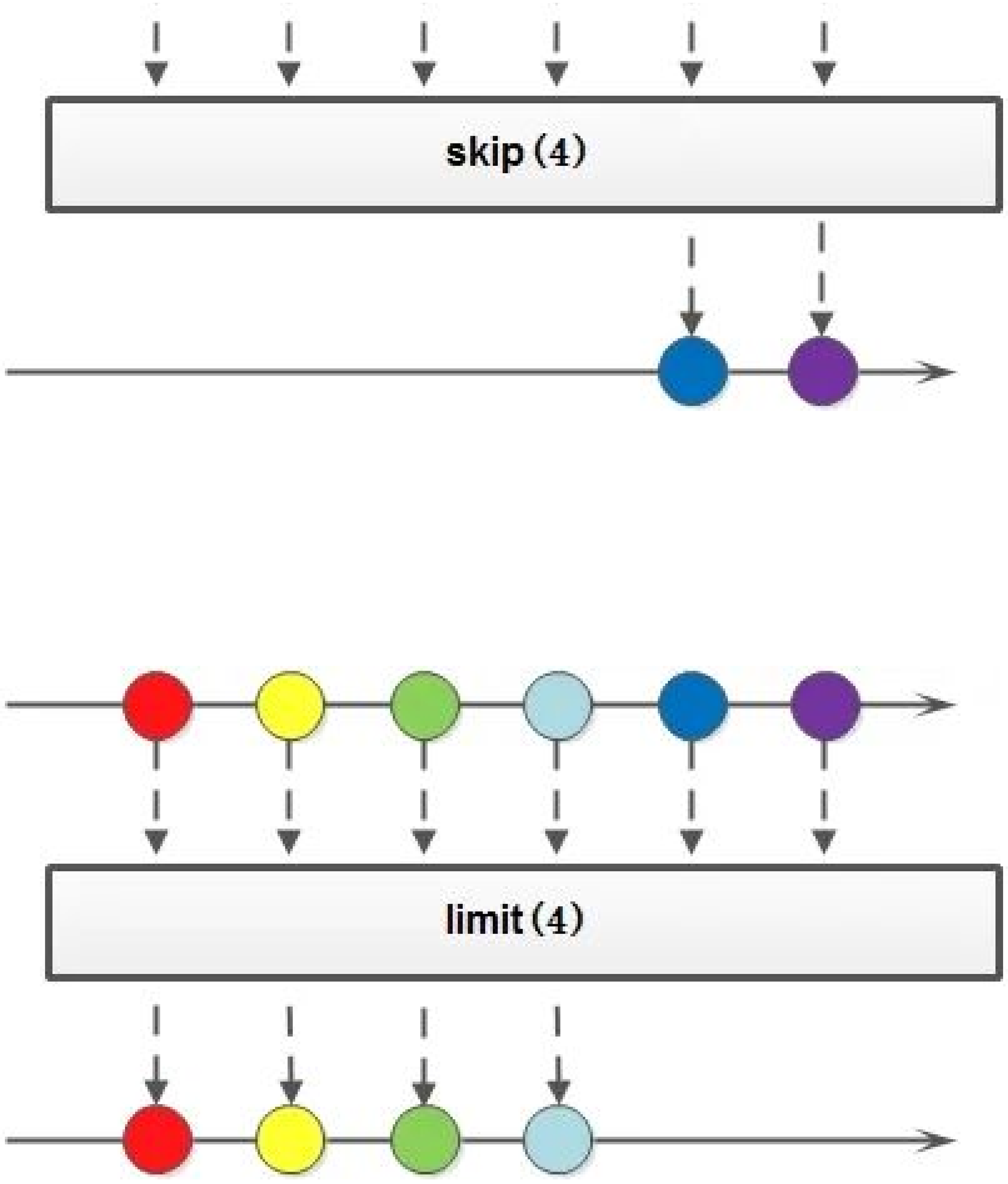
> 按工资升序排序: [Lily, Tom, Sherry, Jack, Alisa]
> 按工资降序排序: [Sherry, Jack, Alisa, Tom, Lily]

> 先按工资再按年龄升序排序： [Lily, Tom, Sherry, Jack, Alisa]
> 先按工资再按年龄自定义降序排序: [Alisa, Jack, Sherry, Tom, Lily]

3.8 提取/组合

流也可以进行合并、去重、限制、跳过等操作。






```
public class StreamTest {
    public static void main(String[] args)
        String[] arr1 = { "a", "b", "c", "d"
        String[] arr2 = { "d", "e", "f", "g"
```

```
        Stream<String> stream1 = Stream.of(a
        Stream<String> stream2 = Stream.of(a
        // concat:合并两个流 distinct: 去重
        List<String> newList = Stream.concat
        // limit: 限制从流中获得前n个数据
        List<Integer> collect = Stream.itera
        // skip: 跳过前n个数据
        List<Integer> collect2 = Stream.iter

        System.out.println("流合并: " + newLi
        System.out.println("limit: " + colle
        System.out.println("skip: " + collec
    }
}
```

运行结果:

> 流合并: [a, b, c, d, e, f, g]
>
> limit: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>
> skip: [3, 5, 7, 9, 11]