什么是Stream

Java8中有两大最为重要的特性。

1)Lambda 表达式，已经学习过了

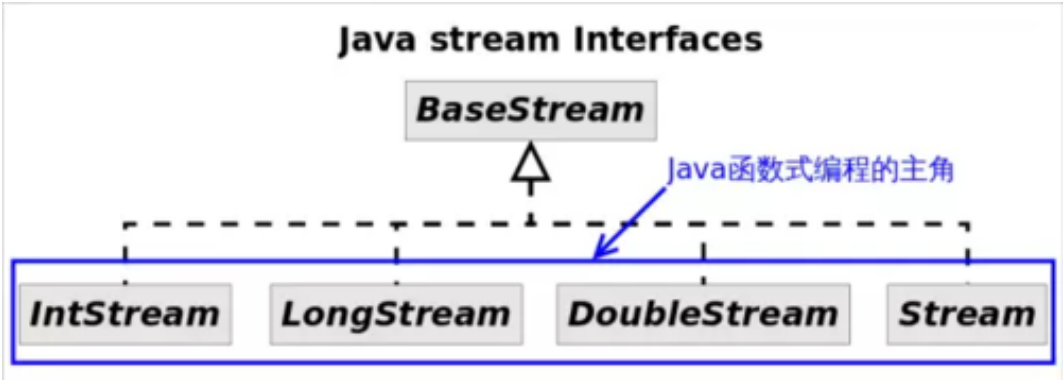
2)Stream API (java.util.stream.*包下)

说到Stream便容易想到I/O Stream，而实际上我们这里讲的Stream它是Java8中对数据处理的一种抽象描述；
我们可以把它理解为数据的管道，我们可以通过这条管道提供给我们的API很方便的对里面的数据进行复杂的操作！比如查找、过滤和映射（类似于使用SQL）；
更厉害的是可以使用Stream API 来并行执行操作;

简而言之，Stream API 提供了一种高效且易于使用的处理数据的方式，解决了已有集合类库操作上的弊端。

总结:Stream是操作数据的一套工具,封装了对数据的各种操作:比如查找、过滤和映射(解决了原有的集合类的弊端)

●注意:
请暂时忘记对传统IO流的固有印象!
Stream接口继承关系



图中

4种stream接口继承自BaseStream，其中IntStream, LongStream, DoubleStream对应三种基本类型（int, long, double，注意不是包装类型），Stream对应所有剩余类型的stream视图。为不同数据类型设置不同stream接口，可以提高性能，并针对不同数据类型提供不同方法实现。那什么不把IntStream等设计成Stream的子接口？毕竟这接口中的方法名大部分是一样的。答案是这些方法的名字虽然相同，但是返回类型不同，如果设计成父子接口关系，这些方法将不能共存，因为Java不允许只有返回类型不同的方法重载。

传统集合操作vs Stream API操作

1.传统方式遍历集合

几乎所有的集合（如Collection 接口或Map 接口等）都支持直接或间接的遍历操作。而当我们需要对集合中的元素进行操作的时候，除了必需的添加、删除、获取外，最典型的就是集合遍历。例如:

```java
 List<String> list = new ArrayList<>();
        list.add("马云");
        list.add("马化腾");
        list.add("李彦宏");
        list.add("雷军");
        list.add("刘强东");
        for (String name : list) {
            System.out.println(name);
        }
```

这是一段非常简单的集合遍历操作：对集合中的每一个字符串都进行打印输出操作。

2.循环遍历的弊端

Java 8的Lambda让我们可以更加专注于做什么（What），而不是怎么做（How），这点此前已经结合匿名内部类进行了对比说明。现在，我们仔细体会一下上例代码，可以发现：
for循环的语法就是“怎么做”
for循环的循环体才是“做什么”(这才是我们要完成的目标！)

如果觉得上面那样的for循环也无所谓的话，我们来完成几个需求：

1)找出姓马的；
2)再从姓马的中找到名字长度等于3的。

如何实现？传统的做法可能为：

```
public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("马云");
        list.add("马化腾");
        list.add("李彦宏");
        list.add("雷军");
        list.add("刘强东");

        // 找出姓马的人
        List<String> newList = new ArrayList<>();
        for (String s : list) {
            if (s.startsWith("马")) {
                newList.add(s);
            }
        }
        for (String name : newList) {
            System.out.println(name);
        }

        System.out.println("----------------");
```

```
        // 在从姓马的人中，找到名字长度等于3的人信息
        List<String> newList1 = new ArrayList<>();
        for (String s : newList) {
            if (s.trim().length()==3) {
                newList1.add(s);
            }
        }
        for (String name : newList1) {
            System.out.println(name);
        }
    }
```

这段代码中含有三个循环，每一个作用不同：

1)首先筛选所有姓张的人；

2)然后筛选名字有三个字的人；

3)最后进行对结果进行打印输出。

每当我们需要对集合中的元素进行操作的时候，总是需要进行循环、循环、再循环。这是理所当然的么？不是。循环是做事情的方式，而不是目的。另一方面，使用线性循环就意味着只能遍历一次。如果希望再次遍历，只能再使用另一个循环从头开始。那Stream能给我们带来怎样更加优雅的写法呢?

3.Stream的优雅写法

下面来看一下如果使用Java 8的Stream API来实现将有多么的优雅：

使用Stream API将我们真正想要做的事情直接体现在代码中。直接阅读代码的字面意思即可完美展示我们要做的事：
获取流、过滤出姓马的、过滤出名字长度为3的、逐一打印。
代码如此简洁，直观，优雅!

```
public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("马云");
        list.add("马化腾");
        list.add("李彦宏");
        list.add("雷军");
        list.add("刘强东");

        // 找出姓马的人
        list.stream().filter(x->x.startsWith("马")).forEach(System.out::println);


        System.out.println("----------------");
        // 在从姓马的人中，找到名字长度等于3的人信息
        list.stream().filter(x->x.startsWith("马"))
                     .filter(x->x.length()==3)
```

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

Stream流

那么，流到底是什么呢？简短的定义就是“从支持数据处理操作的源生成的元素序列”。

●元素序列

——就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。因为集合是数据结构，所以它的主要目的是以特定的时间/空间复杂度存储和访问元素（如ArrayList 与 LinkedList）。但流的目的在于表达计算，比如filter、sorted和map。集合讲的是数据(存储)，流讲的是计算(处理)。

●源

——流会使用一个提供数据的源，如集合、数组或输入/输出资源。 请注意，从有序集合生成流时会保留原有的顺序。由列表生成的流，其元素顺序与列表一致。

●数据处理操作

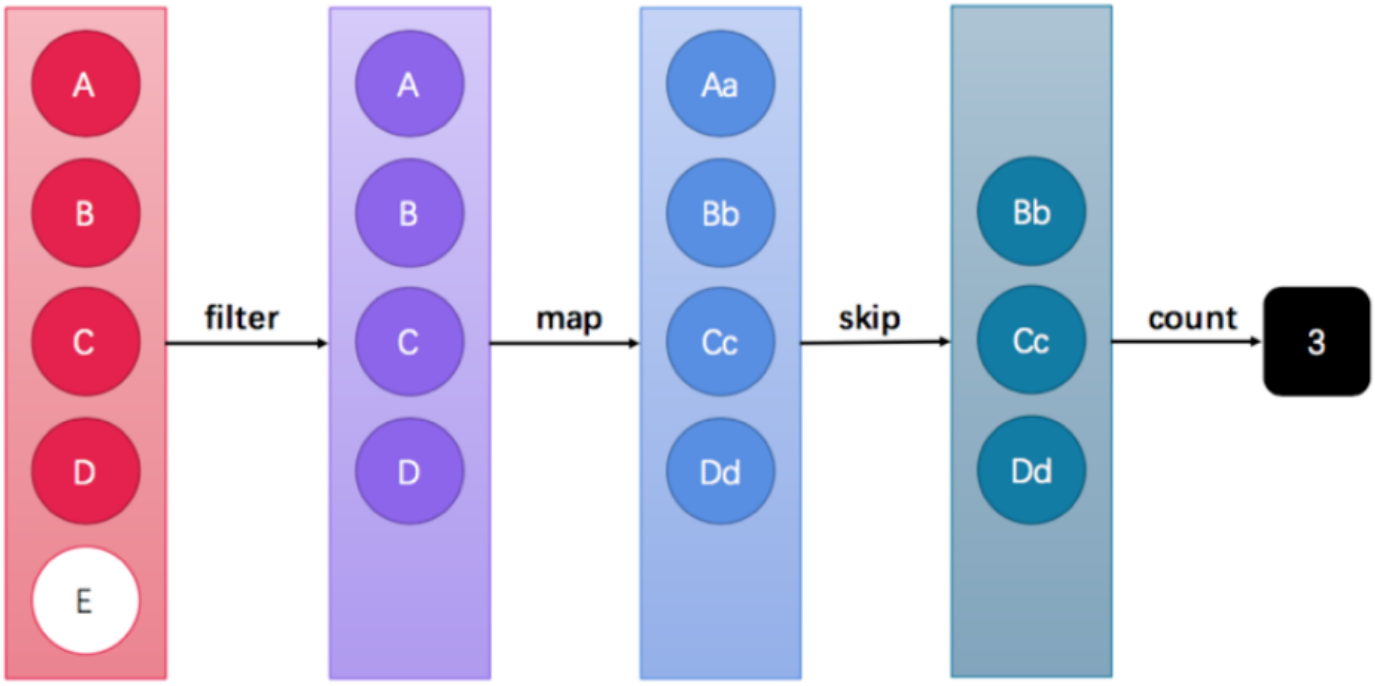
——流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match、sort等。流操作可以顺序执行，也可并行执行。

此外，流操作有两个重要的特点。

> 流水线——很多流操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的流水线。流水线的操作可以看作对数据源进行数据库式查询。

> 内部迭代——与使用迭代器外部迭代的集合不同，流的迭代操作是在背后进行的。

图中展示了过滤、映射、跳过、计数等多步操作，每一个竖着的方框都是一个“流”，调用指定的方法，可以从一个“流”转换为另一个“流”。



Stream流的创建

创建一个流非常简单，有以下几种常用的方式：

1)Collection的默认方法stream()和parallelStream()

2)Arrays.stream()

3)Stream.of()

4)Stream.iterate() （了解）

5)Stream.generate() （了解）

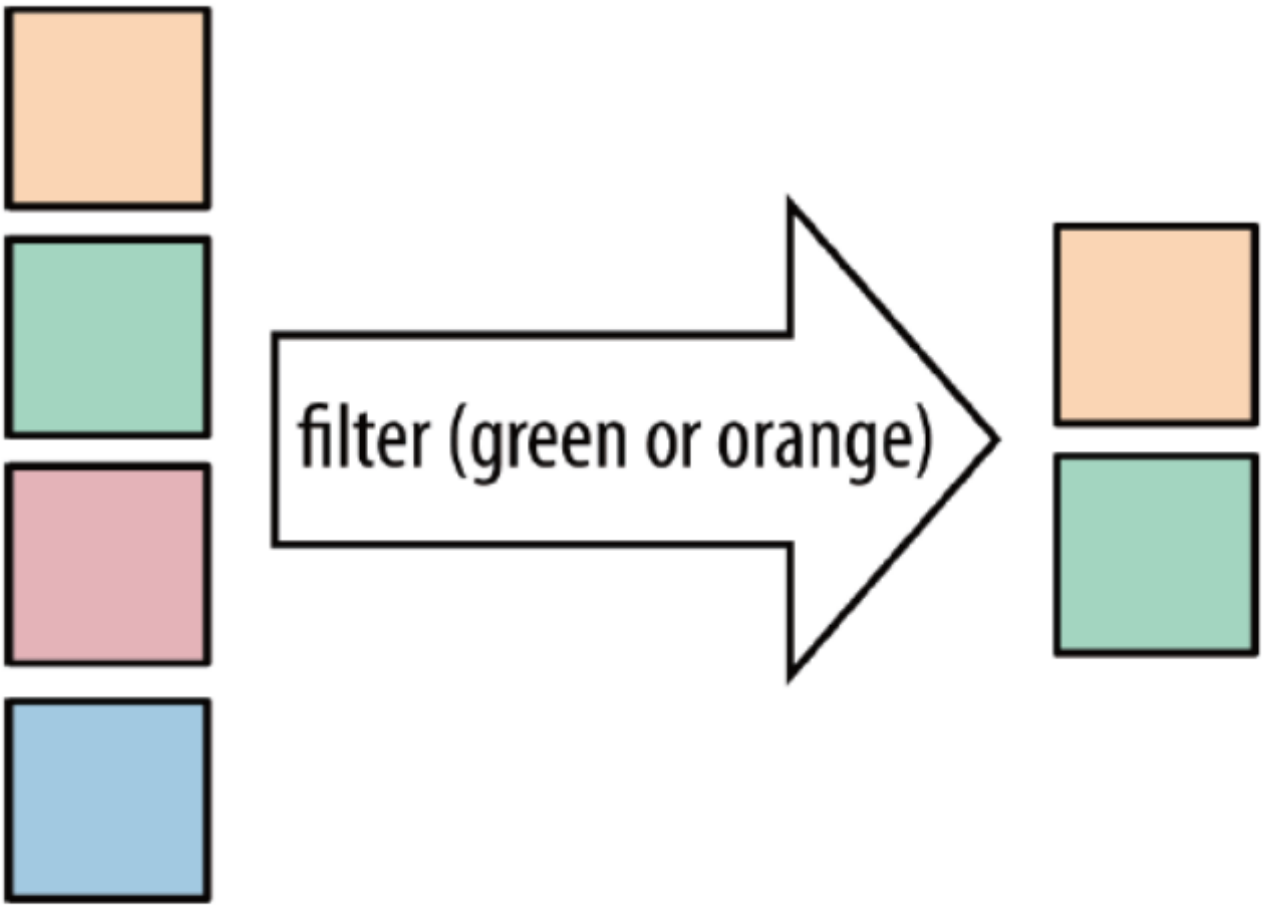
```
public static void main(String[] args) {

        // stream方法和parallelStream方法的区别

        List<Integer> list = Arrays.asList( 4, 3, 2, 4);

        list.stream().forEach(System.out::println);

        System.out.println("----------------------");
```

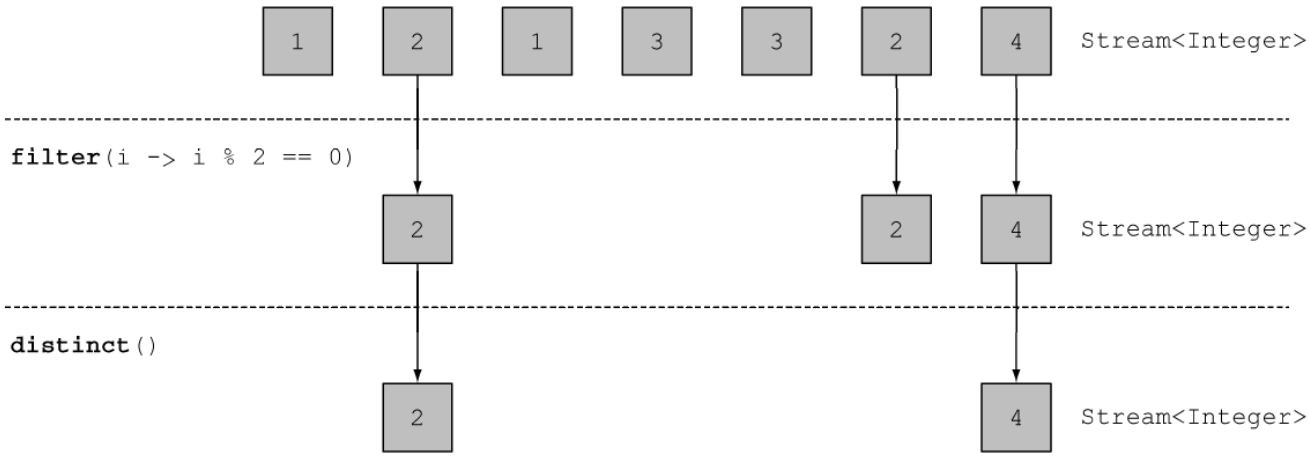
```
        list.parallelStream().forEach(System.out::println);

        System.out.println("----------------------");

        // Arrays.Stream()
        Stream<Integer> arrayOfstream = Arrays.stream(new Integer[]{1, 2, 3, 4});
        arrayOfstream.forEach(System.out::println);


        System.out.println("----------------------");

        Stream<String> integerStream = Stream.of("振兴","中国","未来");
        integerStream.forEach(System.out::println);
        System.out.println("----------------------");

        // Stream.iterate()迭代无限流 (
        // 第一个参数：表示从哪开始，第二个参数：表示以递增2的方式，产生无限流)
        Stream.iterate(10,x->x+2).limit(10).forEach(System.out::println);
        System.out.println("----------------------");

        // Stream.generate() 生成无限流
        // Math::random 表示随机产生 0-1 之间的随机数
        Stream.generate(Math::random).limit(10).forEach(System.out::println);
        System.out.println("----------------------");
        // new Random().nextInt(10) 表示随机产生 1-10 之间的随机数
        Stream.generate(()->new Random().nextInt(10)).limit(10).forEach(System.out::println);

    }
```

Stream 提供的API

筛选和切片

```
filter(Predicate<T> p)：过滤(根据传入的Lambda返回的ture/false 从流中过滤掉某些数据(筛选出某些数据))
 distinct()：去重(根据流中数据的 hashCode和 equals去除重复元素)
 limit(long n)：限定保留n个数据
 skip(long n)：跳过n个数据
```






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

```
System.out.println(2);
System.out.println(4);
```

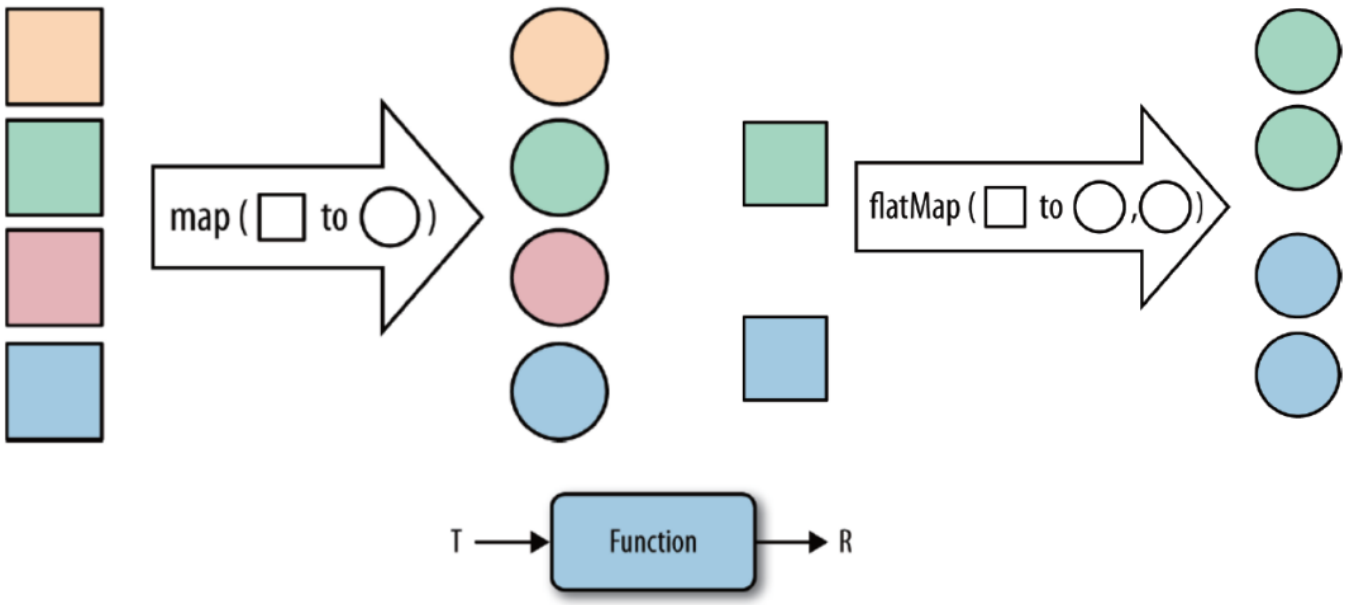
void

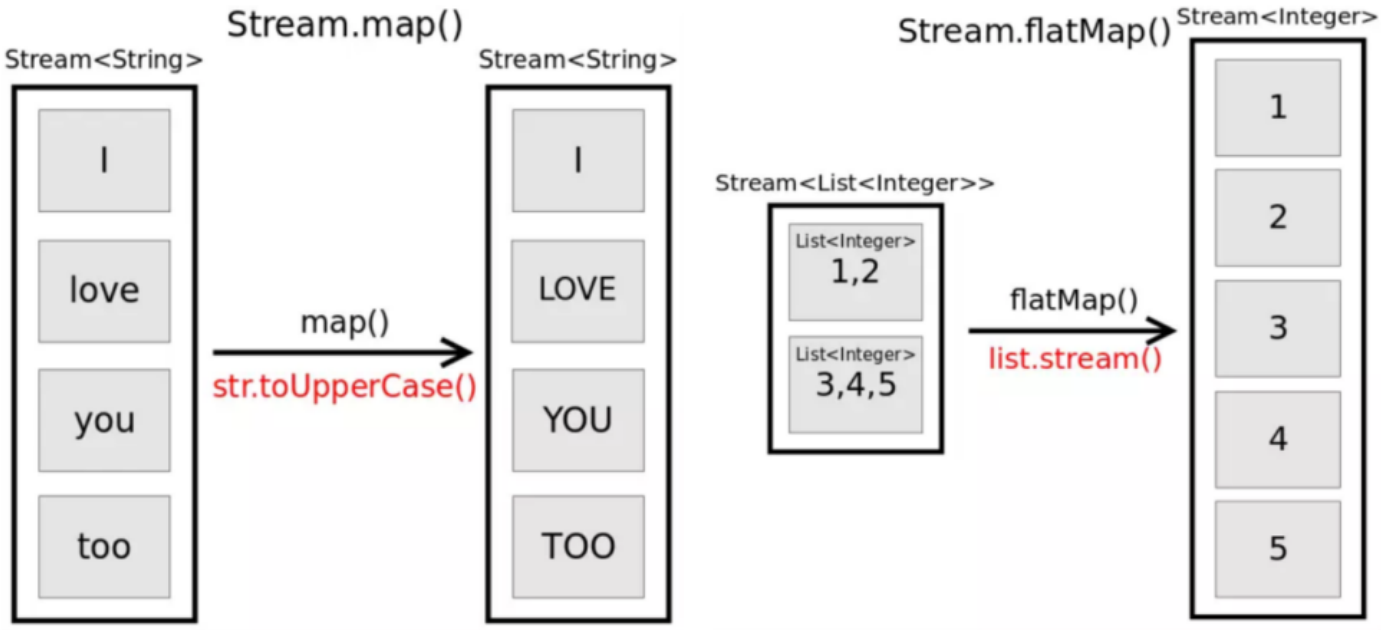
```
public static void main(String[] args) {


        List<Integer> list = Arrays.asList(1, 2, 1, 3, 3, 2, 4);

        //  过滤出偶数并去重复
        list.stream().filter(i->i % 2==0).distinct().forEach(System.out::println);

        System.out.println("----------------------------");
        //  limit(long n)：限定保留n个数据
        list.stream().limit(3).forEach(System.out::println);

        System.out.println("----------------------------");
        //  skip(long n)：跳过n个数据
        list.stream().skip(3).forEach(System.out::println);
        System.out.println("----------------------------");

    }
```

映射

map(Function<T, R> f)：接收一个函数作为参数，该函数会被应用到流中的每个元素上，并将其映射成一个新的元素。

flatMap(Function<T, Stream> mapper)：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






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

        // 需求：把集合中的数据都变成大写
        List<String> list = Arrays.asList("i", "love", "money");
        //list.stream().map(s->s.toUpperCase()).forEach(System.out::println);

        System.out.println("----------------------");
        // 需求：把多个集合合并并取出

        List<Integer> list1 = Arrays.asList(1, 2, 3, 1);
        List<Integer> list2 = Arrays.asList(3, 1, 4, 5);

        Stream<List<Integer>> listStream = Stream.of(list1, list2);
        listStream.flatMap(l->l.stream()).distinct().forEach(System.out::println);
        System.out.println("----------------------");

    }
```

排序

sorted()：自然排序使用Comparable的int compareTo(T o)方法
sorted(Comparator com)：定制排序使用Comparator的int compare(T o1, T o2)方法

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

        List<Integer> list = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
        //自然排序 （升序）
```

```
        list.stream().sorted().forEach(System.out::println);

        System.out.println("----------------------------------");
        // 指定排序规则（降序）
        // 方式一
        list.stream().sorted((o1,o2)->o2.compareTo(o1)).forEach(System.out::println);
        // 方式二
        list.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println);

    }
```

查找匹配

allMatch:检查是否匹配所有元素
anyMatch:检查是否至少匹配一个元素
noneMatch:检查是否没有匹配的元素
findFirst:返回第一个元素(返回值为Optional)
findAny:返回当前流中的任意元素(一般用于并行流)

Optional是Java8新加入的一个容器，这个容器只存1个或0个元素，它用于防止出现NullpointException，它提供如下方法：

•isPresent()

判断容器中是否有值。

•ifPresent(Consume lambda)

容器若不为空则执行括号中的Lambda表达式。

•T get()

获取容器中的元素，若容器为空则抛出NoSuchElement异常。

•T orElse(T other)

获取容器中的元素，若容器为空则返回括号中的默认值。

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

        List<Integer> list = Arrays.asList(1, 2, 1, 3, -1, 2, 4);
//        allMatch:检查是否匹配所有元素
        boolean allMatch = list.stream().allMatch(x -> x > 0);
        System.out.println("检查是否每一个元素都是大于0的:" + allMatch);
//        anyMatch:检查是否至少匹配一个元素
        boolean anyMatch = list.stream().anyMatch(x -> x < 0);
        System.out.println("检查是否存在一个元素是小于0的:" + anyMatch);
//        noneMatch:检查是否没有匹配的元素
        boolean noneMatch = list.stream().noneMatch(x -> x == 0);
```

```
        System.out.println("检查是否真的没有匹配到为0的元素:" + noneMatch);
//        findFirst:返回第一个元素(返回值为Optional<T>)
        Optional<Integer> first = list.stream().findFirst();
        if (first.isPresent()) {
            System.out.println("返回第一个元素:" + first.get());
        }
//        findAny:返回当前流中的任意元素(一般用于并行流)
        Optional<Integer> anyEle = list.stream().findAny();
        System.out.println(anyEle.get());
}
```

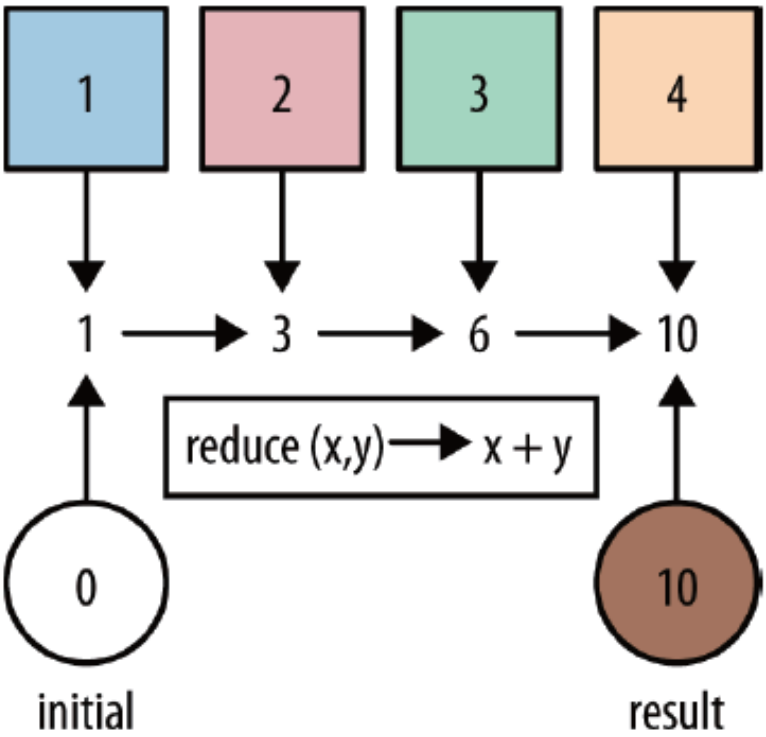
统计

count():返回流中元素的总个数
max(Comparator):返回流中最大值
min(Comparator):返回流中最小值

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

        List<Integer> list = Arrays.asList(1, 2, 1, 3, -1, 2, 4);
//        count():返回流中元素的总个数
        long count = list.stream().count();
        System.out.println("流中的元素的总数: " + count);
//        max(Comparator<T>):返回流中最大值
        Optional<Integer> max = list.stream().max(Comparator.comparing(Integer::intValue));
        System.out.println("流中的元素的最大值: " + max.get());
//        min(Comparator<T>):返回流中最小值
        Optional<Integer> min = list.stream().min(Comparator.comparing(Integer::intValue));
        System.out.println("流中的元素的最小值: " + min.get());

    }
```

归约

reduce(T identity, BinaryOperator) / reduce(BinaryOperator) :将流中元素挨个结合起来，得到一个值。




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

        // 需求:  获取1 到 100 到和
        Integer reduce = Stream.iterate(1, x -> x + 1).limit(100).reduce(0, (x, y) -> x + y);

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

汇总

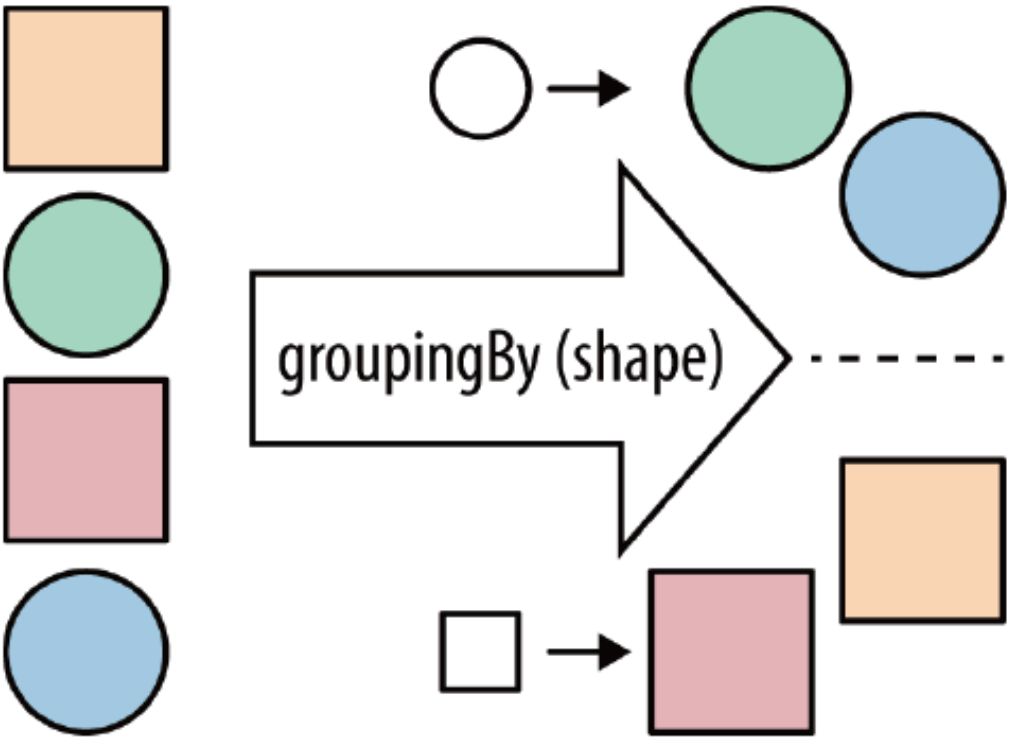
reduce擅长的是生成一个值，如果想要从Stream生成一个集合或者Map等复杂的对象该怎么办呢？终极武器collect()横空出世！
collect(Collector<T, A, R>):将流转换为其他形式。

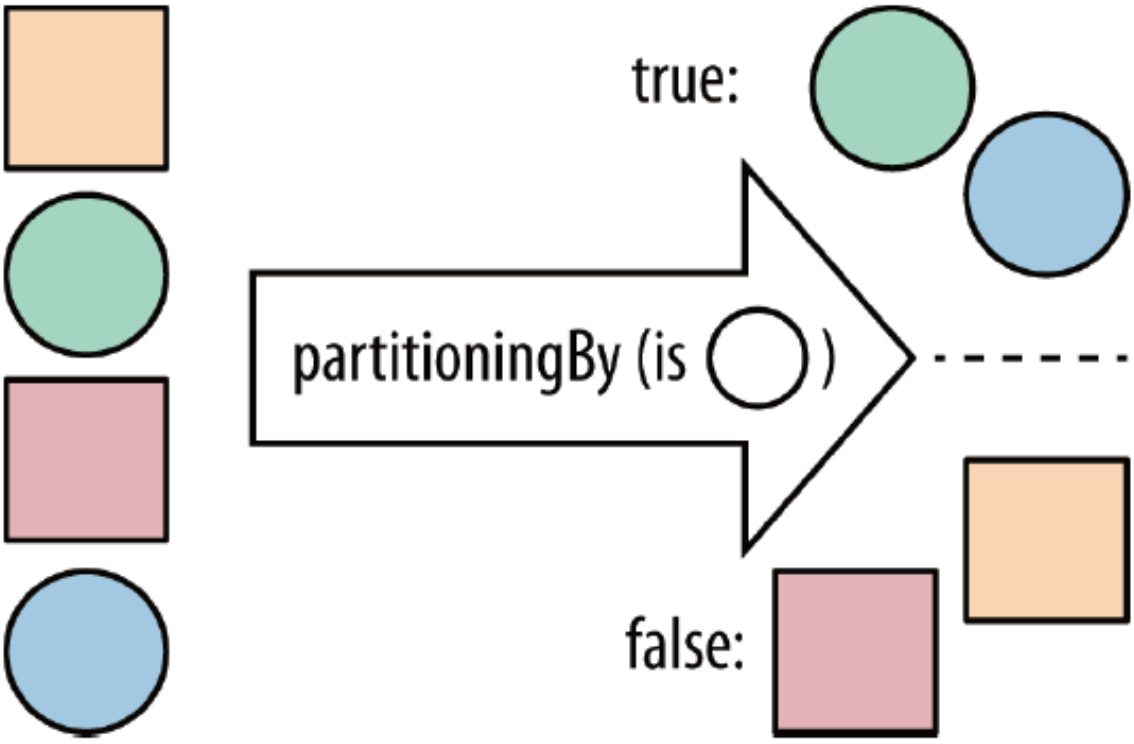
需求:
collect:将流转换为其他形式:list
collect:将流转换为其他形式:set
collect:将流转换为其他形式:TreeSet
collect:将流转换为其他形式:map
collect:将流转换为其他形式:sum
collect:将流转换为其他形式:avg
collect:将流转换为其他形式:max
collect:将流转换为其他形式:min

```
  public static void main(String[] args) {
        List<Integer> streamList = Arrays.asList(1, 2, 1, 3, -1, 2, 4);

//        collect:将流转换为其他形式:list
        List<Integer> integerList = streamList.stream().collect(Collectors.toList());
        System.out.println(integerList.getClass());
        integerList.forEach(System.out::println);

//        collect:将流转换为其他形式:set
        Set<Integer> integerSet = streamList.stream().collect(Collectors.toSet());
        System.out.println(integerSet.getClass());
        integerSet.forEach(System.out::println);


//        collect:将流转换为其他形式:TreeSet
        TreeSet<Integer> integerTreeSet =
streamList.stream().collect(Collectors.toCollection(()-> new TreeSet<>()));
        System.out.println(integerTreeSet.getClass());
        integerTreeSet.forEach(System.out::println);

//        collect:将流转换为其他形式:map (不能去掉重复的数据)
        Map<Integer, Integer> collect =
streamList.stream().distinct().collect(Collectors.toMap(k -> k, v -> v));
        System.out.println(collect.getClass());
        collect.forEach((k,v)->System.out.println(k + ":" + v));


//        collect:将流转换为其他形式:sum
        IntSummaryStatistics summarizing =
streamList.stream().collect(Collectors.summarizingInt(Integer::intValue));
        System.out.println("集合中的数据之和: " + summarizing.getSum());

//        collect:将流转换为其他形式:avg
        Double avg =
streamList.stream().collect(Collectors.averagingDouble(Integer::doubleValue));
        System.out.println("集合中的数据之平均值: "+avg);

//        collect:将流转换为其他形式:max
        Optional<Integer> maxInteger =
streamList.stream().collect(Collectors.maxBy(Integer::compareTo));
        System.out.println("最大值: " + maxInteger.get());

//        collect:将流转换为其他形式:min
        Optional<Integer> minInteger =
streamList.stream().collect(Collectors.minBy(Integer::compareTo));
        System.out.println("最小值: " + minInteger.get());
  }
```

分组和分区

Collectors.groupingBy()对元素做group操作。分组--根据条件分成多个组
Collectors.partitioningBy()对元素进行二分区操作。分区--根据boolean条件分成两个区






```
private static List<Product> products = new ArrayList<>();
```

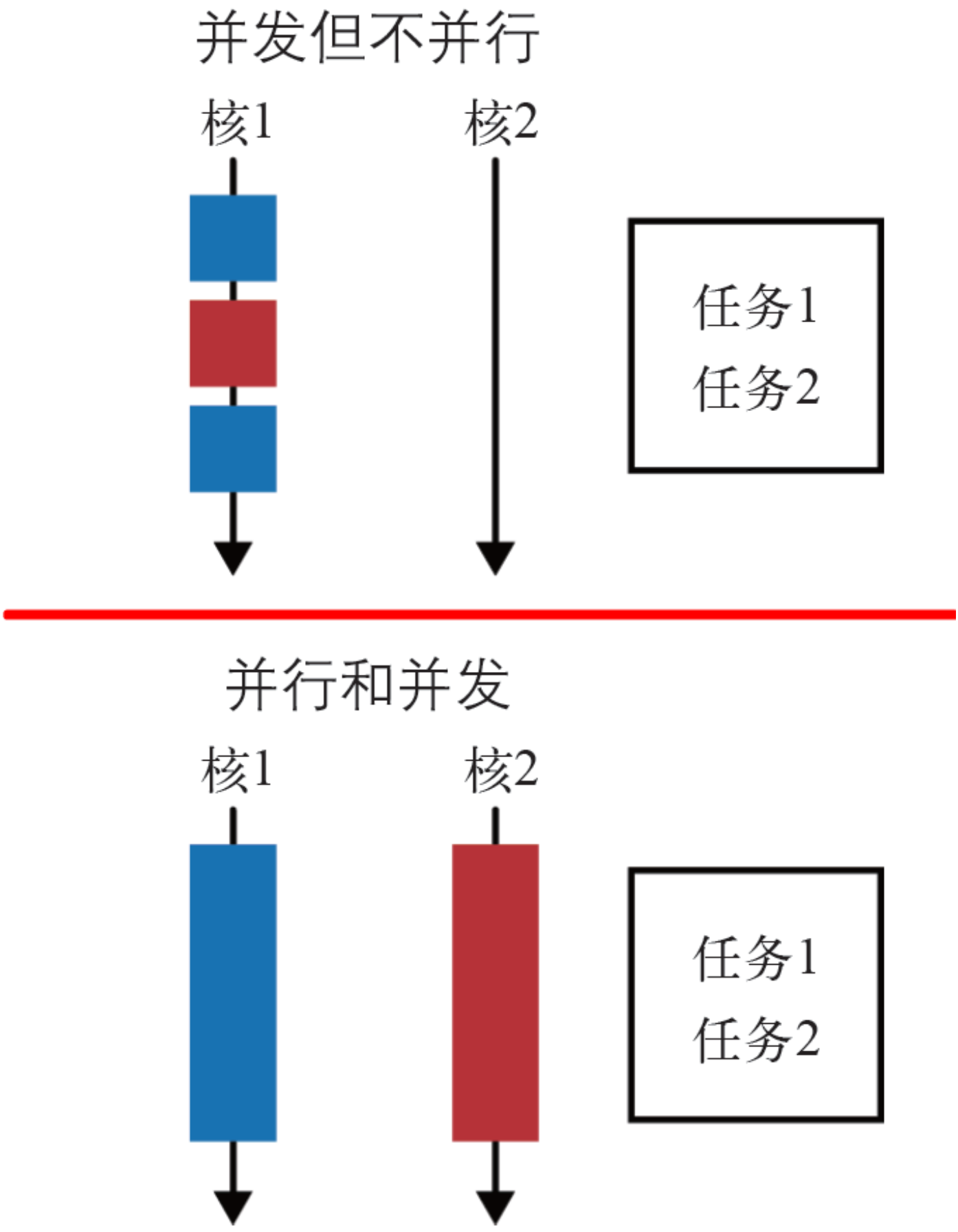
```
     static {
        products.add(new Product(1L, "苹果手机", 8888.88,"手机"));
        products.add(new Product(2L, "华为手机", 6666.66,"手机"));
        products.add(new Product(3L, "联想笔记本", 7777.77,"电脑"));
        products.add(new Product(4L, "机械键盘", 999.99,"键盘"));
        products.add(new Product(5L, "雷蛇鼠标", 222.22,"鼠标"));
    }


    public static void main(String[] args) {

//        根据商品分类名称进行分组
        Map<String, List<Product>> stringListMap =
products.stream().collect(Collectors.groupingBy(Product::getBrand));
        System.out.println(stringListMap);

        System.out.println("-----------------------");

//        根据商品价格是否大于1000进行分区
        Map<Boolean, List<Product>> collect =
products.stream().collect(Collectors.partitioningBy(p -> p.getPrice() > 1000));
        System.out.println(collect);


    }
```

并发和并行

什么是并行和并发

并发是多个任务共享时间段(由CPU切换执行，就像是在同时执行)
并行是多个任务发生在同一时刻(真真正正的同时执行)(必须在多核CPU下)

并行就像用更多的马车（CPU）来拉货（执行任务），货物总量（任务量）一定，那么花费的时间自然减少了。
所以并行可以缩短任务执行时间，提高多核CPU的利用率


并发但不并行
核1
核2
任务1
任务2
并行和并发
核1
核2
任务1
任务2

数据并行化

任务可以并行化执行，同样的数据也可以并行化处理！
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。也就是并行流

并行流

引入

在Java8之前，当需要对存在于集合或数组中的若干元素进行并发操作时，简直就是噩梦！我们需要仔细考虑多线程环境下的原子性、竞争甚至锁问题。
使用Java5的java.util.concurrent.*并发库也还是要考虑诸多细节必须十分谨慎，
使用Java7的fork/join框架编码和调试对于一般程序员来说难度还是太大

而这一切对于Java8中的Stream，不过是小菜一碟！直接调用API方法就可以搞定！
（Stream API 可以声明性地通过parallel() 与sequential() 在并行流与顺序流之间进行切换）

1.转换为并行流

Stream 的父接口java.util.stream.BaseStream 中定义了一个parallel 方法：
只需要在流上调用一下无参数的parallel 方法，那么当前流即可变身成为支持并发操作的流，返回值仍然为Stream 类型。例如：
Stream stream = Stream.of(10, 20, 30, 40, 50).parallel();

2.直接获取并行流

在通过集合获取流时，也可以直接调用parallelStream 方法来直接获取支持并发操作的流。
代码为：
Stream stream = new ArrayList().parallelStream();

3.使用并行流

并行流后续操作的使用方式还是和以前一样，只是底层执行的时候会使用多CPU并行执行
比如多次执行下面这段代码，并行流的输出顺序在很大概率上是不一定的：

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

        // 普通流
        Stream<Integer> integerStream = Stream.of(1, 2, 3, 4);

//        integerStream.forEach(System.out::println);
        System.out.println("---------------------------");
        // 转换成并行流
        Stream<Integer> parallel = integerStream.parallel();
//        parallel.forEach(System.out::println);
        System.out.println("---------------------------");
```

```
        // 转换成普通流
        Stream<Integer> sequential = parallel.sequential();
//        sequential.forEach(System.out::println);

        // 直接转换成并行流
        Stream<Integer> parallelStream = Arrays.asList(1, 2, 3, 4).parallelStream();
        parallelStream.forEach(System.out::println);

    }
```

并行流的效率

●影响并行流性能的主要因素

▲数据大小

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。

▲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。

▲装箱

处理基本类型比处理装箱类型要快。

▲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。

▲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。

●根据性能的好坏，将核心类库提供的通用数据结构分成以下3 组

▲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地被任意分解。

▲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。

▲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Streams.iterate 和BufferedReader.lines，它们长度未知，因此很难预测该在哪里分解。