1. Stream

   Stream表示元素序列，并支持对这些元素进行不同类型的计算操作。Stream操作包含中间操作和终端操作，中间操作返回Stream，终端操作返回void或者一个非Stream的结果值

2. 获得Stream

   可以从各种数据源创建Stream，特别是Collection

   - 从对象list上调用stream()返回一个常规Stream

     ```
     Arrays.asList("a", "b", "c").stream().findFirst().ifPresent(System.out::println);
     // a
     ```

   - 从一堆对象引用中创建一个Stream

     ```
     Stream.of("a", "b", "c").findFirst().ifPresent(System.out::println);
     // a
     ```

   - IntStream获得元素为Int类型的Stream(LongStream, DoubleStream类似)

     ```
     System.out.println(IntStream.of(1, 2, 3).findFirst().getAsInt());
     // 1
     ```

   - 所有这些原生Stream都像普通对象Stream一样工作，但有以下不同：原生Stream使用专门的lambda表达式，例如是IntFunction而不是Function，是IntPredicate，而不是Predicate。原生 Stream 支持额外的终端聚合操作sum()和average()

     ```
     Arrays.stream(new int[] {1, 2, 3})
     .map(n -> 2 * n + 1)
     .average()
     .ifPresent(System.out::println);
     // 5.0

     // 等同于
     IntStream.range(1, 4)
     .map(n -> 2 * n + 1)
     .average()
     .ifPresent(System.out::println);
     // 5.0
     ```

   - 将普通Stream转换成原生Stream

     ```
     Stream.of("a1", "a2", "a3")
     .map(s -> s.substring(1))
     .mapToInt(Integer::parseInt)
     .max()
     .ifPresent(System.out::println);
     // 3
     ```

   - 将原生Stream转换成普通Stream

```
    IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);
    // a1
    // a2
    // a3
```

3. 处理顺序

中间操作的一个重要特征式惰性，如果终端操作缺失，中间操作不会执行

```
Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> {
                System.out.println("filter: " + s);
                return true;
            });
// 控制台不会有输出
```

```
 Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> {
                System.out.println("filter: " + s);
                return true;
            })
            .forEach(s -> System.out.println("forEach: " + s));
// filter: d2
// forEach: d2
// filter: a2
// forEach: a2
// filter: b1
// forEach: b1
// filter: b3
// forEach: b3
// filter: c
// forEach: c
```

输出顺序可能令人惊讶。一种简单的方法是在Stream的所有元素上水平地执行操作。但此处相反，每个元素都沿着链垂直移动。第一个字符串“d2”先filter然后foreach，然后第二个字符串“a2”才被处理。

这种方式可以减少在每个元素上执行的实际操作数，如下例所示:

```
Stream.of("d2", "a2", "b1", "b3", "c")
.map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
})
.anyMatch(s -> {
    System.out.println("anyMatch: " + s);
    return s.startsWith("A");
});

// map:      d2
// anyMatch: D2
// map:      a2
// anyMatch: A2
```

当predicate应用于给定的输入元素时，anyMatch将立即返回true。这对于第二个被传递的“A2”来说是正确的。由于stream链的垂直执行，在这种情况下，map只会执行两次。因此，map将尽可能少地被调用，而不是所有的元素映射到Stream中。

4. 为什么处理顺序很重要

```
Stream.of("d2", "a2", "b1", "b3", "c")
.map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
})
.filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("A");
})
.forEach(s -> System.out.println("forEach: " + s));

// map:     d2
// filter:  D2
// map:     a2
// filter:  A2
// forEach: A2
// map:     b1
// filter:  B1
// map:     b3
// filter:  B3
// map:     c
// filter:  C
```

如果我们改变操作的顺序，将filter移到链的开头，我们可以大大减少实际执行次数:

```
Stream.of("d2", "a2", "b1", "b3", "c")
.filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("a");
})
.map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
})
.forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

扩展

```
Stream.of("d2", "a2", "b1", "b3", "c")
.sorted((s1, s2) -> {
    System.out.printf("sort: %s; %s\n", s1, s2);
    return s1.compareTo(s2);
})
.filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("a");
})
.map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
})
.forEach(s -> System.out.println("forEach: " + s));
```

排序是一种特殊的中间操作。这是所谓的状态操作，因为要对元素进行排序，你需要维护元素的状态。

执行此示例将在控制台输出:

```
sort:    a2; d2
sort:    b1; a2
sort:    b1; d2
sort:    b1; a2
sort:    b3; b1
sort:    b3; d2
sort:    c; b3
sort:    c; d2
filter:  a2
map:     a2
forEach: A2
filter:  b1
filter:  b3
filter:  c
filter:  d2
```

首先，在整个输入集合上执行排序操作。换句话说， sorted是水平执行的。因此，在这个例子中，对输入集合中的每个元素进行多次组合， sorted被调用8次

我们再一次通过对链操作重排序来优化性能:

```
Stream.of("d2", "a2", "b1", "b3", "c")
.filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("a");
})
.sorted((s1, s2) -> {
    System.out.printf("sort: %s; %s\n", s1, s2);
    return s1.compareTo(s2);
})
.map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
})
.forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// filter:  b1
// filter:  b3
// filter:  c
// map:     a2
// forEach: A2
```

在这个示例中，没有调用 sorted，因为filter将输入集合减少到一个元素。因此，对于大数据量的输入集合，性能会极大地提高

5. Stream复用

Java 8 Stream 无法复用。一旦你调用任何终端操作， Stream就会关闭

```
  Stream<String> stream =
            Stream.of("d2", "a2", "b1", "b3", "c")
                    .filter(s -> s.startsWith("a"));
stream.anyMatch(s -&gt; true);
// ok
stream.noneMatch(s -&gt; true);
// exception
```

为了克服这个限制，必须为要执行的每一个终端操作创建一个新的Stream链，例如，我们可以创建一个Stream提供者来创建已构建所有中间操作的新Stream

```
Supplier<Stream<String>> streamSupplier =
() -> Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);
// ok
streamSupplier.get().noneMatch(s -> true);
// ok
```

6. 高级操作collect()

Collect是一种非常有用的终端操作，可以将stream元素转换为不同类型的结果，例如List, Set or Map。Collect 接受一个包含四个不同操作的Collector: supplier, accumulator, combiner 和 finisher。这听起来很复杂，优点是Java 8通过Collectors类支持各种内置收集器。因此，对于最常见的操作，你不必自己实现Collector。

```
 List<String> filtered =
            Arrays.asList("a1", "a2", "c")
                    .stream()
                    .filter(p -> p.startsWith("a"))
                    .collect(Collectors.toList());
    System.out.println(filtered);
// [a1, a2]
```

如果需要set而不是list 使用Collectors.toSet()就可以。

分组

```
Map<Integer, List<String>> group =
            Arrays.asList("a1", "a2", "c")
                    .stream()
                    .collect(Collectors.groupingBy(String::length));
    System.out.println(group);
// {1=[c], 2=[a1, a2]}
```