

Полное руководство по Java 8 Stream API в картинках и примерах

9.08.2017 / 14:35 от **aNNiMON** (/ablogs/?act=user&id=1)

Java (/articles/java) java 8 (/str/search.php?category=article-tags&q=java%208), stream api (/str/search.php?category=article-tags&q=stream%20api), collector (/str/search.php?category=article-tags&q=collector), flatmap (/str/search.php?category=article-tags&q=flatmap), interactive (/str/search.php?category=article-tags&q=interactive)

С момента выхода Java 8 я практически сразу начал пользоваться Stream API, так как функциональный подход обработки данных мне пришелся по нраву. Хотелось пользоваться им везде, поэтому я начал разрабатывать библиотеку [Lightweight-Stream-API](https://github.com/aNNiMON/Lightweight-Stream-API) (<https://github.com/aNNiMON/Lightweight-Stream-API>), которая привносит подобный подход в ранние версии Java. Также меня интересовало внутреннее устройство стримов. За это время накопилось достаточно опыта и теперь я спешу им поделиться.

В статье, помимо описания стримов, приводятся визуальные демонстрации работы операторов, примеры и задачи для самопроверки. Также затронуты нововведения, касающиеся стримов в Java 9.

Дабы не путать стримы (Stream) с I/O потоками (InputStream/OutputStream) и тредами/нитьями/потоками (Thread), я буду придерживаться англоязычного именования в транслите и называть Stream стримом.

Содержание

(javascript:showspoiler('a246','b246','c246')) (javascript:showspoiler('a246','b246','c246'))

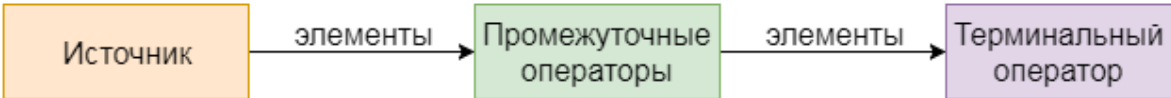
Обновления:

2021-05-08: Добавлены описания [mapMulti \(#mapMulti\)](#) и [toList \(#toList\)](#) из Java 16

1. Stream

Stream — это объект для универсальной работы с данными. Мы указываем, какие операции хотим провести, при этом не заботясь о деталях реализации. Например, *взять элементы из списка сотрудников, выбрать тех, кто младше 40 лет, отсортировать по фамилии и поместить в новый список*. Или чуть сложнее, *прочитать все json-файлы, находящиеся в папке books, десериализовать в список объектов книг, обработать элементы всех этих списков, а затем сгруппировать книги по автору*.

Данные могут быть получены из источников, коими являются коллекции или методы, поставляющие данные. Например, список файлов, массив строк, метод range() для числовых промежутков и т.д. То есть, стрим использует существующие коллекции для получения новых элементов, это ни в коем случае не новая структура данных. К данным затем применяются операторы. Например, взять лишь некоторые элементы (filter), преобразовать каждый элемент (map), посчитать сумму элементов или объединить всё в один объект (reduce).



Операторы можно разделить на две группы:

- Промежуточные (intermediate) — обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.
- Терминальные (terminal) — обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.

2. Получение объекта Stream

Пока что хватит теории. Пришло время посмотреть, как создать или получить объект `java.util.stream.Stream`.

- Пустой стрим: `Stream.empty()` // `Stream<String>`
- Стрим из List: `list.stream()` // `Stream<String>`
- Стрим из Map: `map.entrySet().stream()` // `Stream<Map.Entry<String, String>>`
- Стрим из массива: `Arrays.stream(array)` // `Stream<String>`

Поделиться

Похожие статьи

- Halik - Java-отладчик нового поколения (/article/2290)
- GameLib / Урок 2 (Вторая версия GL, создание окна) (/article/2550)
- java.util.concurrent ScheduledThreadPoolExecutor как замена классу Timer (/article/3184)
- Графическое поле ввода (/article/1827)
- Дизайн API библиотеки (/article/256)

Другие статьи автора

- Как написать игру под Android за 15 дней. История создания Mega Flood-It. Часть первая (/article/3473)
- Кодировки (/article/48)
- Таймер в Java ME (/article/305)
- Делаем Eclipse похожим на Netbeans (/article/3)
- Посмотреть все (106 статей) (/ablogs/?act=user&id=1)

Дорогой посетитель

Реклама позволяет нашему ресурсу существовать и развиваться. Пожалуйста, отключите блокировщик рекламы или зарегистрируйтесь (/registration.php).
Спасибо. Приятного чтения. 🍪

- Стрим из указанных элементов: `Stream.of("a", "b", "c") // Stream<String>`

А вот и пример:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. public static void main(String[] args) {
2.     List<String> list = Arrays.stream(args)
3.         .filter(s -> s.length() <= 2)
4.         .collect(Collectors.toList());
5. }
```

В данном примере источником служит метод `Arrays.stream`, который из массива `args` делает стрим. Промежуточный оператор `filter` отбирает только те строки, длина которых не превышает два. Терминальный оператор `collect` собирает полученные элементы в новый список.

И ещё один пример:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. IntStream.of(120, 410, 85, 32, 314, 12)
2.     .filter(x -> x < 300)
3.     .map(x -> x + 11)
4.     .limit(3)
5.     .forEach(System.out::print)
```

Здесь уже три промежуточных оператора:

- `filter` — отбирает элементы, значение которых меньше 300,
- `map` — прибавляет 11 к каждому числу,
- `limit` — ограничивает количество элементов до 3.

Терминальный оператор `forEach` применяет функцию `print` к каждому приходящему элементу.



На ранних версиях Java этот пример выглядел бы так:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. int[] arr = {120, 410, 85, 32, 314, 12};
2. int count = 0;
3. for (int x : arr) {
4.     if (x >= 300) continue;
5.     x += 11;
6.     count++;
7.     if (count > 3) break;
8.     System.out.print(x);
9. }
```

С увеличением числа операторов код в ранних версиях усложнялся бы на порядок, не говоря уже о том, что разбить вычисления на несколько потоков при таком подходе было бы крайне непросто.

3. Как работает стрим

У стримов есть некоторые особенности. Во-первых, обработка не начнётся до тех пор, пока не будет вызван терминальный оператор. `list.stream().filter(x -> x > 100)`; не возьмёт ни единого элемента из списка. Во-вторых, стрим после обработки нельзя переиспользовать.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Stream<String> stream = list.stream();
2. stream.forEach(System.out::println);
3. stream.filter(s -> s.contains("Stream API"));
4. stream.forEach(System.out::println);
```

Код на второй строке выполнится, а вот на третьей выбросит исключение

```
java.lang.IllegalStateException: stream has already been operated upon or closed.
```

Исходя из первой особенности, делаем вывод, что обработка происходит от терминального оператора к источнику. Это действительно так и это удобно. Мы можем в качестве источника использовать генерируемую бесконечную последовательность, скажем, факториала или чисел Фибоначчи, но обрабатывать лишь некоторую её часть.



Пока мы не присоединили терминальный оператор, доступа к источнику не проводилось. Как только появился терминальный оператор `forEach`, он стал запрашивать элементы у стоящего перед ним оператора `limit`. Тот в свою очередь обращается к `map`, `map` к `filter`, а `filter` уже обращается к источнику. Затем элементы поступают в прямом порядке: источник, `filter`, `map`, `limit` и `forEach`. Пока какой-либо из операторов не обработает элемент должным образом, новые запрошены не будут. Как только через оператор `limit` прошло 3 элемента, он переходит в закрытое состояние и больше не будет запрашивать элементы у `map`. `forEach` запрашивает очередной элемент, но `limit` сообщает, что больше не может поставить элементов, поэтому `forEach` делает вывод, что элементы закончились и прекращает работу.

Такой подход зовётся `pull iteration`, то есть элементы запрашиваются у источника по мере надобности. К слову, в `RxJava` реализован `push iteration` подход, то есть источник сам уведомляет, что появились элементы и их нужно обработать.

4. Параллельные стримы

Стримы бывают последовательными (`sequential`) и параллельными (`parallel`). Последовательные выполняются только в текущем потоке, а вот параллельные используют общий пул `ForkJoinPool.commonPool()` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool-->). При этом элементы разбиваются (если это возможно) на несколько групп и обрабатываются в каждом потоке отдельно. Затем на нужном этапе группы объединяются в одну для предоставления конечного результата.

Чтобы получить параллельный стрим, нужно либо вызвать метод `parallelStream()` вместо `stream()`, либо превратить обычный стрим в параллельный, вызвав промежуточный оператор `parallel`.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. list.parallelStream()
2.   .filter(x -> x > 10)
3.   .map(x -> x * 2)
4.   .collect(Collectors.toList());
5.
6. IntStream.range(0, 10)
7.   .parallel()
8.   .map(x -> x * 10)
9.   .sum();
```

Работа с потокобезопасными коллекциями, разбиение элементов на части, создание потоков, объединение частей воедино, всё это кроется в реализации `Stream API`. От нас лишь требуется вызвать нужный метод и проследить, чтобы функции в операторах не зависели от каких-либо внешних факторов, иначе есть риск получить неверный результат или ошибку.

Вот так делать нельзя:

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	<code>final List<Integer> ints = new ArrayList<>();</code>
2.	<code>IntStream.range(0, 1000000)</code>
3.	<code>.parallel()</code>
4.	<code>.forEach(i -> ints.add(i));</code>
5.	<code>System.out.println(ints.size());</code>

Это код Шрёдингера. Он может нормально выполниться и показать 1000000, может выполниться и показать 869877, а может и упасть с ошибкой `Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 332 at java.util.ArrayList.add(ArrayList.java:459)`.

Поэтому разработчики настоятельно просят воздержаться от побочных эффектов в лямбдах, то тут, то там говоря в документации о *невмешательстве* (*non-interference*).

5. Стримы для примитивов

Кроме объектных стримов `Stream<T>`, существуют специальные стримы для примитивных типов:

- `IntStream` для `int`,
- `LongStream` для `long`,
- `DoubleStream` для `double`.

Для `boolean`, `byte`, `short` и `char` специальных стримов не придумали, но вместо них можно использовать `IntStream`, а затем приводить к нужному типу. Для `float` тоже придётся воспользоваться `DoubleStream`.

Примитивные стримы полезны, так как не нужно тратить время на боксинг/анбоксинг, к тому же у них есть ряд специальных операторов, упрощающих жизнь. Их мы рассмотрим очень скоро.

6. Операторы Stream API

Дальше приводятся операторы Stream API с описанием, демонстрацией и примерами. Можете использовать это как справочник.

6.1. Источники

`empty()`

Стрим, как и коллекция, может быть пустым, а значит всем последующем операторам нечего будет обрабатывать.

1.	<code>Stream.empty()</code>
2.	<code>.forEach(System.out::println);</code>
3.	<code>// Вывода нет</code>

`of(T value)`

`of(T... values)`

Стрим для одного или нескольких перечисленных элементов. Очень часто вижу, что используют такую конструкцию:

1.	<code>Arrays.asList(1, 2, 3).stream()</code>
2.	<code>.forEach(System.out::println);</code>

однако она излишня. Вот так проще:

1.	<code>Stream.of(1, 2, 3)</code>
2.	<code>.forEach(System.out::println);</code>

`ofNullable(T t)`

Появился в Java 9. Возвращает пустой стрим, если в качестве аргумента передан `null`, в противном случае, возвращает стрим из одного элемента.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>String str = Math.random() > 0.5 ? "I'm feeling lucky" : null;</code>
2.	<code>Stream.ofNullable(str)</code>
3.	<code>.forEach(System.out::println);</code>

generate(Supplier s)

Возвращает стрим с бесконечной последовательностью элементов, генерируемых функцией Supplier s.

```
1. Stream.generate(() -> 6)
2.   .limit(6)
3.   .forEach(System.out::println);
4. // 6, 6, 6, 6, 6, 6
```



Поскольку стрим бесконечный, нужно его ограничивать или осторожно использовать, дабы не попасть в бесконечный цикл.

iterate(T seed, UnaryOperator f)

Возвращает бесконечный стрим с элементами, которые образуются в результате последовательного применения функции f к итерируемому значению. Первым элементом будет seed, затем f(seed), затем f(f(seed)) и так далее.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Stream.iterate(2, x -> x + 6)
2.   .limit(6)
3.   .forEach(System.out::println);
4. // 2, 8, 14, 20, 26, 32
```



[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Stream.iterate(1, x -> x * 2)
2.   .limit(6)
3.   .forEach(System.out::println);
4. // 1, 2, ____, ____, ____, 32
```

iterate(T seed, Predicate hasNext, UnaryOperator f)

Появился в Java 9. Всё то же самое, только добавляется ещё один аргумент hasNext: если он возвращает false, то стрим завершается. Это очень похоже на цикл for:

```
1. for (i = seed; hasNext(i); i = f(i)) {
2. }
```

Таким образом, с помощью iterate теперь можно создать конечный стрим.

```
1. Stream.iterate(2, x -> x < 25, x -> x + 6)
2.   .forEach(System.out::println);
3. // 2, 8, 14, 20
```

0:00 / 0:09

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	Stream.iterate(4, x -> x < 100, x -> x * 4)
2.	.forEach(System.out::println);
3.	// _____, 16, _____

concat(Stream a, Stream b)

Объединяет два стрима так, что вначале идут элементы стрима A, а по его окончанию последуют элементы стрима B.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	Stream.concat(
2.	Stream.of(1, 2, 3),
3.	Stream.of(4, 5, 6))
4.	.forEach(System.out::println);
5.	// 1, 2, 3, 4, 5, 6

0:00 / 0:10

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	Stream.concat(
2.	Stream.of(10),
3.	Stream.of(_____, _____))
4.	.forEach(System.out::println);
5.	// 10, 4, 16

builder()

Создаёт мутабельный объект для добавления элементов в стрим без использования какого-либо контейнера для этого.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	Stream.Builder<Integer> streamBuider = Stream.<Integer>builder()
2.	.add(0)
3.	.add(1);
4.	for (int i = 2; i <= 8; i += 2) {
5.	streamBuider.accept(i);
6.	}
7.	streamBuider
8.	.add(9)
9.	.add(10)
10.	.build()
11.	.forEach(System.out::println);
12.	// 0, 1, 2, 4, 6, 8, 9, 10

IntStream.range(int startInclusive, int endExclusive)

LongStream.range(long startInclusive, long endExclusive)

Создаёт стрим из числового промежутка [start..end), то есть от start (включительно) по end.

[\[копировать\]](#) (javascript:void(0);) [\[скачать\]](#) (javascript:void(0);)

```
1. IntStream.range(0, 10)
2.     .forEach(System.out::println);
3. // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
4.
5. LongStream.range(-10L, -5L)
6.     .forEach(System.out::println);
7. // -10, -9, -8, -7, -6
```

IntStream.rangeClosed(int startInclusive, int endInclusive)

LongStream.rangeClosed(long startInclusive, long endInclusive)

Создаёт стрим из числового промежутка [start..end], то есть от start (включительно) по end (включительно).

[\[копировать\]](#) (javascript:void(0);) [\[скачать\]](#) (javascript:void(0);)

```
1. IntStream.rangeClosed(0, 5)
2.     .forEach(System.out::println);
3. // 0, 1, 2, 3, 4, 5
4.
5. LongStream.range(-8L, -5L)
6.     .forEach(System.out::println);
7. // -8, -7, -6, -5
```

6.2. Промежуточные операторы

filter(Predicate predicate)

Фильтрует стрим, принимая только те элементы, которые удовлетворяют заданному условию.

[\[копировать\]](#) (javascript:void(0);) [\[скачать\]](#) (javascript:void(0);)

```
1. Stream.of(1, 2, 3)
2.     .filter(x -> x == 10)
3.     .forEach(System.out::print);
4. // Вывода нет, так как после фильтрации стрим станет пустым
5.
6. Stream.of(120, 410, 85, 32, 314, 12)
7.     .filter(x -> x > 100)
8.     .forEach(System.out::println);
9. // 120, 410, 314
```

0:00 / 0:12

[\[копировать\]](#) (javascript:void(0);) [\[скачать\]](#) (javascript:void(0);)

```
1. IntStream.range(2, 9)
2.     .filter(x -> x % ____ == 0)
3.     .forEach(System.out::println);
4. // 3, 6
```

map(Function mapper)

Применяет функцию к каждому элементу и затем возвращает стрим, в котором элементами будут результаты функции. map можно применять для изменения типа элементов.

Stream.mapToDouble(ToDoubleFunction mapper)

Stream.mapToInt(ToIntFunction mapper)

Stream.mapToLong(ToLongFunction mapper)

IntStream.mapToObj(IntFunction mapper)

IntStream.mapToLong(IntToLongFunction mapper)

IntStream.mapToDouble(IntToDoubleFunction mapper)

Специальные операторы для преобразования объектного стрима в примитивный, примитивного в объектный, либо примитивного стрима одного типа в примитивный стрим другого.

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	Stream.of("3", "4", "5")
2.	.map(Integer::parseInt)
3.	.map(x -> x + 10)
4.	.forEach(System.out::println);
5.	// 13, 14, 15
6.	
7.	Stream.of(120, 410, 85, 32, 314, 12)
8.	.map(x -> x + 11)
9.	.forEach(System.out::println);
10.	// 131, 421, 96, 43, 325, 23

0:00 / 0:12

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	Stream.of("10", "11", " ")
2.	.map(x -> Integer.parseInt(x, 16))
3.	.forEach(System.out::println);
4.	// , , 50

flatMap(Function<T, Stream<R>> mapper)

Один из самых интересных операторов. Работает как map, но с одним отличием — можно преобразовать один элемент в ноль, один или множество других.

```
flatMapToDouble(Function mapper)
flatMapToInt(Function mapper)
flatMapToLong(Function mapper)
```

Как и в случае с map, служат для преобразования в примитивный стрим.

Для того, чтобы один элемент преобразовать в ноль элементов, нужно вернуть null, либо пустой стрим. Чтобы преобразовать в один элемент, нужно вернуть стрим из одного элемента, например, через Stream.of(x). Для возвращения нескольких элементов, можно любыми способами создать стрим с этими элементами.

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	Stream.of(2, 3, 0, 1, 3)
2.	.flatMap(x -> IntStream.range(0, x))
3.	.forEach(System.out::println);
4.	// 0, 1, 0, 1, 2, 0, 0, 1, 2

0:00 / 0:10

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	Stream.of(1, 2, 3, 4, 5, 6)
2.	.flatMap(x -> {


```
3.         switch (x % 2) {
4.             case 0:
5.                 return Stream.of(x, x*x, x*x*2);
6.             case 1:
7.                 return Stream.of(x);
8.             case 2:
9.                 default:
10.                return Stream.empty();
11.         }
12.     })
13.     .forEach(System.out::println);
14. // 1, 3, 9, 18, 4, 6, 36, 72
```

mapMulti(BiConsumer<T, Consumer<R>> mapper)

Появился в Java 16. Этот оператор похож на flatMap, но использует императивный подход при работе. Теперь вместе с элементом стрима приходит ещё и Consumer, в который можно передать одно или несколько значений, либо не передавать вовсе.

Вот как было с flatMap:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Stream.of(1, 2, 3, 4, 5, 6)
2.     .flatMap(x -> {
3.         if (x % 2 == 0) {
4.             return Stream.of(-x, x);
5.         }
6.         return Stream.empty();
7.     })
8.     .forEach(System.out::println);
9. // -2, 2, -4, 4, -6, 6
```

А вот так можно переписать с использованием mapMulti:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Stream.of(1, 2, 3, 4, 5, 6)
2.     .mapMulti((x, consumer) -> {
3.         if (x % 2 == 0) {
4.             consumer.accept(-x);
5.             consumer.accept(x);
6.         }
7.     })
8.     .forEach(System.out::println);
9. // -2, 2, -4, 4, -6, 6
```

mapMultiToDouble(BiConsumer<T, DoubleConsumer> mapper)
mapMultiToInt(BiConsumer<T, IntConsumer> mapper)
mapMultiToLong(BiConsumer<T, LongConsumer> mapper)

Служат для преобразования в примитивный стрим.

У mapMulti есть несколько преимуществ перед flatMap. Во-первых, если приходится пропускать значения (как в примере выше, где пропускались нечётные элементы), то не будет затрат на создание пустого стрима. Во-вторых, consumer легко передать в другой метод, в котором можно проводить преобразования, включая рекурсивные.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. void processSerializable(Serializable ser, Consumer<String> consumer) {
2.     if (ser instanceof String str) {
3.         consumer.accept(str);
4.     } else if (ser instanceof List) {
5.         for (Serializable s : (List<Serializable>) ser) {
6.             processSerializable(s, consumer);
7.         }
8.     }
9. }
10.
11. Serializable arr(Serializable... elements) {
12.     return Arrays.asList(elements);
13. }
14.
15. Stream.of(arr("A", "B"), "C", "D", arr(arr("E"), "F"), "G")
16.     .mapMulti(this::processSerializable)
17.     .forEach(System.out::println);
18. // A, B, C, D, E, F, G
```

limit(long maxSize)

Ограничивает стрим maxSize элементами.

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	Stream.of(120, 410, 85, 32, 314, 12)
2.	.limit(4)
3.	.forEach(System.out::println);
4.	// 120, 410, 85, 32

0:00 / 0:10

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	Stream.of(120, 410, 85, 32, 314, 12)
2.	.limit()
3.	.limit(5)
4.	.forEach(System.out::println);
5.	// 120, 410
6.	
7.	Stream.of(19)
8.	.limit()
9.	.forEach(System.out::println);
10.	// Вывода нет

skip(long n)

Пропускает n элементов стрима.

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	Stream.of(5, 10)
2.	.skip(40)
3.	.forEach(System.out::println);
4.	// Вывода нет
5.	
6.	Stream.of(120, 410, 85, 32, 314, 12)
7.	.skip(2)
8.	.forEach(System.out::println);
9.	// 85, 32, 314, 12

0:00 / 0:10

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

1.	IntStream.range(0, 10)
2.	.limit(5)
3.	.skip(3)
4.	.forEach(System.out::println);
5.	// ,
6.	
7.	IntStream.range(0, 10)
8.	.skip(5)
9.	.limit(3)
10.	.skip(1)
11.	.forEach(System.out::println);

12.

// ,

sorted()

sorted(Comparator comparator)

Сортирует элементы стрима. Причём работает этот оператор очень хитро: если стрим уже помечен как отсортированный, то сортировка проводиться не будет, иначе соберёт все элементы, отсортирует их и вернёт новый стрим, помеченный как отсортированный. См. [9.1 \(#characteristics\)](#).

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.

IntStream.range(0, 100000000)

2.

.sorted()

3.

.limit(3)

4.

.forEach(System.out::println);

5.

// 0, 1, 2

6.

7.

IntStream.concat(

8.

IntStream.range(0, 100000000),

9.

IntStream.of(-1, -2))

10.

.sorted()

11.

.limit(3)

12.

.forEach(System.out::println);

13.

// Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

14.

15.

Stream.of(120, 410, 85, 32, 314, 12)

16.

.sorted()

17.

.forEach(System.out::println);

18.

// 12, 32, 85, 120, 314, 410

0:00 / 0:10

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.

Stream.of(120, 410, 85, 32, 314, 12)

2.

.sorted(Comparator. Order())

3.

.forEach(System.out::println);

4.

// 410, 314, 120, 85, 32, 12

distinct()

Убирает повторяющиеся элементы и возвращаем стрим с уникальными элементами. Как и в случае с sorted, смотрит, состоит ли уже стрим из уникальных элементов и если это не так, отбирает уникальные и помечает стрим как содержащий уникальные элементы.

1.

Stream.of(2, 1, 8, 1, 3, 2)

2.

.distinct()

3.

.forEach(System.out::println);

4.

// 2, 1, 8, 3

0:00 / 0:12

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.

IntStream.concat(

2.

IntStream.range(2, 5),

```
3.      IntStream.range(0, 4))
4.      .distinct()
5.      .forEach(System.out::println);
6.  //  ____, ____, ____, ____, ____
```

peek(Consumer action)

Выполняет действие над каждым элементом стрима и при этом возвращает стрим с элементами исходного стрима. Служит для того, чтобы передать элемент куда-нибудь, не разрывая при этом цепочку операторов (вы же помните, что forEach — терминальный оператор и после него стрим завершается?), либо для отладки.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1.  Stream.of(0, 3, 0, 0, 5)
2.      .peek(x -> System.out.format("before distinct: %d\n", x))
3.      .distinct()
4.      .peek(x -> System.out.format("after distinct: %d\n", x))
5.      .map(x -> x * x)
6.      .forEach(x -> System.out.format("after map: %d\n", x));
7.  // before distinct: 0
8.  // after distinct: 0
9.  // after map: 0
10. // before distinct: 3
11. // after distinct: 3
12. // after map: 9
13. // before distinct: 1
14. // after distinct: 1
15. // after map: 1
16. // before distinct: 5
17. // before distinct: 0
18. // before distinct: 5
19. // after distinct: 5
20. // after map: 25
```

0:00 / 0:12

takeWhile(Predicate predicate)

Появился в Java 9. Возвращает элементы до тех пор, пока они удовлетворяют условию, то есть функция-предикат возвращает true. Это как limit, только не с числом, а с условием.

```
1.  Stream.of(1, 2, 3, 4, 2, 5)
2.      .takeWhile(x -> x < 3)
3.      .forEach(System.out::println);
4.  // 1, 2
```

0:00 / 0:09

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1.  IntStream.range(2, 7)
2.      .takeWhile(x -> x != ____ )
3.      .forEach(System.out::println);
4.  // 2, 3, 4
```

dropWhile(Predicate predicate)

Появился в Java 9. Пропускает элементы до тех пор, пока они удовлетворяют условию, затем возвращает оставшуюся часть стрима. Если предикат вернул для первого элемента false, то ни единого элемента не будет пропущено. Оператор подобен skip, только работает по условию.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Stream.of(1, 2, 3, 4, 2, 5)
2.     .dropWhile(x -> x >= 3)
3.     .forEach(System.out::println);
4. // 1, 2, 3, 4, 2, 5
5.
6. Stream.of(1, 2, 3, 4, 2, 5)
7.     .dropWhile(x -> x < 3)
8.     .forEach(System.out::println);
9. // 3, 4, 2, 5
```



[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. IntStream.range(2, 7)
2.     .dropWhile(x -> x < ____ )
3.     .forEach(System.out::println);
4. // 5, 6
5.
6. IntStream.of(1, 3, 2, 0, 5, 4)
7.     .dropWhile(x -> x ____ 2 == 1)
8.     .forEach(System.out::println);
9. // 2, 0, 5, 6
```

boxed()

Преобразует примитивный стрим в объектный.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. DoubleStream.of(0.1, Math.PI)
2.     .boxed()
3.     .map(Object::getClass)
4.     .forEach(System.out::println);
5. // class java.lang.Double
6. // class java.lang.Double
```

6.3. Терминальные операторы

void forEach(Consumer action)

Выполняет указанное действие для каждого элемента стрима.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Stream.of(120, 410, 85, 32, 314, 12)
2.     .forEach(x -> System.out.format("%s, ", x));
3. // 120, 410, 85, 32, 314, 12
```

void forEachOrdered(Consumer action)

Тоже выполняет указанное действие для каждого элемента стрима, но перед этим добивается правильного порядка вхождения элементов. Используется для параллельных стримов, когда нужно получить правильную последовательность элементов.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. IntStream.range(0, 100000)
2.     .parallel()
3.     .filter(x -> x % 10000 == 0)
4.     .map(x -> x / 10000)
5.     .forEach(System.out::println);
6. // 5, 6, 7, 3, 4, 8, 0, 9, 1, 2
7.
8. IntStream.range(0, 100000)
9.     .parallel()
10.    .filter(x -> x % 10000 == 0)
11.    .map(x -> x / 10000)
12.    .forEachOrdered(System.out::println);
13. // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

long count()

Возвращает количество элементов стрима.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. long count = IntStream.range(0, 10)
2.     .flatMap(x -> IntStream.range(0, x))
3.     .count();
4. System.out.println(count);
5. // 45
6.
7. System.out.println(
8.     IntStream.rangeClosed(-3, ____ )
9.     .count()
10. );
11. // 10
12.
13. System.out.println(
14.     Stream.of(0, 2, 9, 13, 5, 11)
15.         . ____ (x -> x ____ 2)
16.         .filter(x -> x % 2 == 1)
17.         .count()
18. );
19. // 0
```

R collect(Collector collector)

Один из самых мощных операторов Stream API. С его помощью можно собрать все элементы в список, множество или другую коллекцию, сгруппировать элементы по какому-нибудь критерию, объединить всё в строку и т.д.. В классе java.util.stream.Collectors очень много методов на все случаи жизни, мы рассмотрим их позже. При желании можно [написать свой коллектор \(#collector-implementation\)](#), реализовав интерфейс Collector.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. List<Integer> list = Stream.of(1, 2, 3)
2.     .collect(Collectors.toList());
3. // list: [1, 2, 3]
4.
5. String s = Stream.of(1, 2, 3)
6.     .map(String::valueOf)
7.     .collect(Collectors.joining("-", "<", ">"));
8. // s: "<1-2-3>"
```

R collect(Supplier supplier, BiConsumer accumulator, BiConsumer combiner)

То же, что и collect(collector), только параметры разбиты для удобства. Если нужно быстро сделать какую-то операцию, нет нужды реализовывать интерфейс Collector, достаточно передать три лямбда-выражения.

supplier должен поставлять новые объекты (контейнеры), например new ArrayList(), accumulator добавляет элемент в контейнер, combiner необходим для параллельных стримов и объединяет части стрима воедино.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	List<String> list = Stream.of("a", "b", "c", "d")
2.	.collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
3.	// List: ["a", "b", "c", "d"]

Object[] toArray()

Возвращает нетипизированный массив с элементами стрима.

A[] toArray(IntFunction<A[]> generator)
Аналогично, только возвращает типизированный массив.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	String[] elements = Stream.of("a", "b", "c", "d")
2.	.toArray(String[]::new);
3.	// elements: ["a", "b", "c", "d"]

List<T> toList()

Наконец-то добавлен в Java 16. Возвращает список, подобно collect(Collectors.toList()). Отличие в том, что теперь возвращаемый список гарантированно нельзя будет модифицировать. Любое добавление или удаление элементов в полученный список будет сопровождаться исключением UnsupportedOperationException.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

1.	List<String> elements = Stream.of("a", "b", "c", "d")
2.	.map(String::toUpperCase)
3.	.toList();
4.	// elements: ["A", "B", "C", "D"]

T reduce(T identity, BinaryOperator accumulator)
U reduce(U identity, BiFunction accumulator, BinaryOperator combiner)

Ещё один полезный оператор. Позволяет преобразовать все элементы стрима в один объект. Например, посчитать сумму всех элементов, либо найти минимальный элемент.

Сперва берётся объект identity и первый элемент стрима, применяется функция accumulator и identity становится её результатом. Затем всё продолжается для остальных элементов.

1.	int sum = Stream.of(1, 2, 3, 4, 5)
2.	.reduce(10, (acc, x) -> acc + x);
3.	// sum: 25

Optional reduce(BinaryOperator accumulator)



Этот метод отличается тем, что у него нет начального объекта identity. В качестве него служит первый элемент стрима. Поскольку стрим может быть пустой и тогда identity объект не присвоится, то результатом функции служит Optional, позволяющий обработать и эту ситуацию, вернув Optional.empty().

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Optional<Integer> result = Stream.<Integer>empty()
2.   .reduce((acc, x) -> acc + x);
3. System.out.println(result.isPresent());
4. // false
5.
6. Optional<Integer> sum = Stream.of(1, 2, 3, 4, 5)
7.   .reduce((acc, x) -> acc + x);
8. System.out.println(sum.get());
9. // 15
```

0:00 / 0:09

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. int sum = IntStream.of(2, 4, 6, 8)
2.   .reduce(____, (acc, x) -> acc + x);
3. // sum: 25
4.
5. int product = IntStream.range(0, 10)
6.   .filter(x -> x++ % 4 == 0)
7.   .reduce((acc, x) -> acc * x)
8.   .getAsInt();
9. // product: _____
```

Optional min(Comparator comparator)

Optional max(Comparator comparator)

Поиск минимального/максимального элемента, основываясь на переданном компараторе. Внутри вызывается reduce:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. reduce((a, b) -> comparator.compare(a, b) <= 0 ? a : b));
2. reduce((a, b) -> comparator.compare(a, b) >= 0 ? a : b));
```

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. int min = Stream.of(20, 11, 45, 78, 13)
2.   .min(Integer::compare).get();
3. // min: 11
4.
5. int max = Stream.of(20, 11, 45, 78, 13)
6.   .max(Integer::compare).get();
7. // max: 78
```

Optional findAny()

Возвращает первый попавшийся элемент стрима. В параллельных стримах это может быть действительно любой элемент, который лежал в разбитой части последовательности.

Optional findFirst()

Гарантированно возвращает первый элемент стрима, даже если стрим параллельный.

Если нужен любой элемент, то для параллельных стримов быстрее будет работать findAny().

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. int anySeq = IntStream.range(4, 65536)
2.   .findAny()
3.   .getAsInt();
4. // anySeq: 4
```

```
5.
6.  int firstSeq = IntStream.range(4, 65536)
7.    .findFirst()
8.    .getAsInt();
9.  // firstSeq: 4
10.
11.  int anyParallel = IntStream.range(4, 65536)
12.    .parallel()
13.    .findAny()
14.    .getAsInt();
15.  // anyParallel: 32770
16.
17.  int firstParallel = IntStream.range(4, 65536)
18.    .parallel()
19.    .findFirst()
20.    .getAsInt();
21.  // firstParallel: 4
```

boolean allMatch(Predicate predicate)

Возвращает true, если все элементы стрима удовлетворяют условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет false, то оператор перестаёт просматривать элементы и возвращает false.

```
1.  boolean result = Stream.of(1, 2, 3, 4, 5)
2.    .allMatch(x -> x <= 7);
3.  // result: true
```



```
1.  boolean result = Stream.of(1, 2, 3, 4, 5)
2.    .allMatch(x -> x < 3);
3.  // result: false
```



[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1.  boolean result = Stream.of(120, 410, 85, 32, 314, 12)
2.    .allMatch(x -> x % 2 == 0);
3.  // result: _____
```

boolean anyMatch(Predicate predicate)

Возвращает true, если хотя бы один элемент стрима удовлетворяет условию predicate. Если такой элемент встретился, нет смысла продолжать перебор элементов, поэтому сразу возвращается результат.

```
1.  boolean result = Stream.of(1, 2, 3, 4, 5)
2.    .anyMatch(x -> x == 3);
3.  // result: true
```

0:00 / 0:06

1.	<code>boolean</code> result = Stream.of(1, 2, 3, 4, 5)
2.	<code>.anyMatch(x -> x == 8);</code>
3.	<code>// result: false</code>

0:00 / 0:10

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>boolean</code> result = Stream.of(120, 410, 85, 32, 314, 12)
2.	<code>.anyMatch(x -> x % 22 == 0);</code>
3.	<code>// result: _____</code>

boolean noneMatch(Predicate predicate)

Возвращает true, если, пройдя все элементы стрима, ни один не удовлетворил условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет true, то оператор перестаёт перебирать элементы и возвращает false.

1.	<code>boolean</code> result = Stream.of(1, 2, 3, 4, 5)
2.	<code>.noneMatch(x -> x == 9);</code>
3.	<code>// result: true</code>

0:00 / 0:10

1.	<code>boolean</code> result = Stream.of(1, 2, 3, 4, 5)
2.	<code>.noneMatch(x -> x == 3);</code>
3.	<code>// result: false</code>

0:00 / 0:06

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>boolean</code> result = Stream.of(120, 410, 86, 32, 314, 12)
2.	<code>.noneMatch(x -> x % 2 == 1);</code>

3.	<code>// result: </code> <u> </u>
----	--------------------------------------------

OptionalDouble average()

Только для примитивных стримов. Возвращает среднее арифметическое всех элементов. Либо Optional.empty, если стрим пуст.

1.	<code>double result = IntStream.range(2, 16)</code>
2.	<code> .average()</code>
3.	<code> .getAsDouble();</code>
4.	<code>// result: 8.5</code>

sum()

Возвращает сумму элементов примитивного стрима. Для IntStream результат будет типа int, для LongStream — long, для DoubleStream — double.

1.	<code>long result = LongStream.range(2, 16)</code>
2.	<code> .sum();</code>
3.	<code>// result: 119</code>

IntSummaryStatistics summaryStatistics()

Полезный метод примитивных стримов. Позволяет собрать статистику о числовой последовательности стрима, а именно: количество элементов, их сумму, среднее арифметическое, минимальный и максимальный элемент.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>LongSummaryStatistics stats = LongStream.range(2, 16)</code>
2.	<code> .summaryStatistics();</code>
3.	<code>System.out.format(" count: %d%n", stats.getCount());</code>
4.	<code>System.out.format(" sum: %d%n", stats.getSum());</code>
5.	<code>System.out.format("average: %.1f%n", stats.getAverage());</code>
6.	<code>System.out.format(" min: %d%n", stats.getMin());</code>
7.	<code>System.out.format(" max: %d%n", stats.getMax());</code>
8.	<code>// count: 14</code>
9.	<code>// sum: 119</code>
10.	<code>// average: 8,5</code>
11.	<code>// min: 2</code>
12.	<code>// max: 15</code>

7. Методы Collectors

toList()

Самый распространённый метод. Собирает элементы в List.

toSet()

Собирает элементы в множество.

toCollection(Supplier collectionFactory)

Собирает элементы в заданную коллекцию. Если нужно конкретно указать, какой List, Set или другую коллекцию мы хотим использовать, то этот метод поможет.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>Deque<Integer> deque = Stream.of(1, 2, 3, 4, 5)</code>
2.	<code> .collect(Collectors.toCollection(ArrayDeque::new));</code>
3.	
4.	<code>Set<Integer> set = Stream.of(1, 2, 3, 4, 5)</code>
5.	<code> .collect(Collectors.toCollection(LinkedHashSet::new));</code>

toMap(Function keyMapper, Function valueMapper)

Собирает элементы в Map. Каждый элемент преобразовывается в ключ и в значение, основываясь на результате функций keyMapper и valueMapper соответственно. Если нужно вернуть тот же элемент, что и пришел, то можно передать Function.identity().

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	

```
1. Map<Integer, Integer> map1 = Stream.of(1, 2, 3, 4, 5)
2.     .collect(Collectors.toMap(
3.         Function.identity(),
4.         Function.identity()
5.     ));
6. // {1=1, 2=2, 3=3, 4=4, 5=5}
7.
8. Map<Integer, String> map2 = Stream.of(1, 2, 3)
9.     .collect(Collectors.toMap(
10.        Function.identity(),
11.        i -> String.format("%d * 2 = %d", i, i * 2)
12.    ));
13. // {1="1 * 2 = 2", 2="2 * 2 = 4", 3="3 * 2 = 6"}
14.
15. Map<Character, String> map3 = Stream.of(50, 54, 55)
16.     .collect(Collectors.toMap(
17.        i -> (char) i.intValue(),
18.        i -> String.format("<%d>", i)
19.    ));
20. // {'2'="<50>", '6'="<54>", '7'="<55>"}
```

toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction)

Аналогичен первой версии метода, только в случае, когда встречается два одинаковых ключа, позволяет объединить значения.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Map<Integer, String> map4 = Stream.of(50, 55, 69, 20, 19, 52)
2.     .collect(Collectors.toMap(
3.        i -> i % 5,
4.        i -> String.format("<%d>", i),
5.        (a, b) -> String.join(", ", a, b)
6.    ));
7. // {0="<50>, <55>, <20>", 2="<52>", 4="<64>, <19>"}
```

В данном случае, для чисел 50, 55 и 20, ключ одинаков и равен 0, поэтому значения накапливаются. Для 64 и 19 аналогично.

toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapFactory)

Всё то же, только позволяет указывать, какой именно класс Map использовать.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Map<Integer, String> map5 = Stream.of(50, 55, 69, 20, 19, 52)
2.     .collect(Collectors.toMap(
3.        i -> i % 5,
4.        i -> String.format("<%d>", i),
5.        (a, b) -> String.join(", ", a, b),
6.        LinkedHashMap::new
7.    ));
8. // {0=<50>, <55>, <20>, 4=<69>, <19>, 2=<52>}
```

Отличие этого примера от предыдущего в том, что теперь сохраняется порядок, благодаря LinkedHashMap.

toConcurrentMap(Function keyMapper, Function valueMapper)

toConcurrentMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction)

toConcurrentMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapFactory)

Всё то же самое, что и toMap, только работаем с ConcurrentMap.

collectingAndThen(Collector downstream, Function finisher)

Собирает элементы с помощью указанного коллектора, а потом применяет к полученному результату функцию.

[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. List<Integer> list = Stream.of(1, 2, 3, 4, 5)
2.     .collect(Collectors.collectingAndThen(
3.        Collectors.toList(),
4.        Collections::unmodifiableList));
5. System.out.println(list.getClass());
6. // class java.util.Collections$UnmodifiableRandomAccessList
7.
8. List<String> list2 = Stream.of("a", "b", "c", "d")
```

9.	.collect(Collectors.collectingAndThen(
10.	Collectors.toMap(Function.identity(), s -> s + s),
11.	map -> map.entrySet().stream()))
12.	.map(e -> e.toString())
13.	.collect(Collectors.collectingAndThen(
14.	Collectors.toList(),
15.	Collections::unmodifiableList));
16.	list2.forEach(System.out::println);
17.	// a=aa
18.	// b=bb
19.	// c=cc
20.	// d=dd

joining()

joining(CharSequence delimiter)

joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

Собирает элементы, реализующие интерфейс CharSequence, в единую строку.

Дополнительно можно указать разделитель, а также префикс и суффикс для всей последовательности.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	String s1 = Stream.of("a", "b", "c", "d")
2.	.collect(Collectors.joining());
3.	System.out.println(s1);
4.	// abcd
5.	
6.	String s2 = Stream.of("a", "b", "c", "d")
7.	.collect(Collectors.joining("-"));
8.	System.out.println(s2);
9.	// a-b-c-d
10.	
11.	String s3 = Stream.of("a", "b", "c", "d")
12.	.collect(Collectors.joining(" -> ", "[", "]"));
13.	System.out.println(s3);
14.	// [a -> b -> c -> d]

summingInt(ToIntFunction mapper)

summingLong(ToLongFunction mapper)

summingDouble(ToDoubleFunction mapper)

Коллектор, который преобразовывает объекты в int/long/double и подсчитывает сумму.

averagingInt(ToIntFunction mapper)

averagingLong(ToLongFunction mapper)

averagingDouble(ToDoubleFunction mapper)

Аналогично, но со средним значением.

summarizingInt(ToIntFunction mapper)

summarizingLong(ToLongFunction mapper)

summarizingDouble(ToDoubleFunction mapper)

Аналогично, но с полной статистикой.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	Integer sum = Stream.of("1", "2", "3", "4")
2.	.collect(Collectors.summingInt(Integer::parseInt));
3.	System.out.println(sum);
4.	// 10
5.	
6.	Double average = Stream.of("1", "2", "3", "4")
7.	.collect(Collectors.averagingInt(Integer::parseInt));
8.	System.out.println(average);
9.	// 2.5
10.	
11.	DoubleSummaryStatistics stats = Stream.of("1.1", "2.34", "3.14", "4.04")
12.	.collect(Collectors.summarizingDouble(Double::parseDouble));
13.	System.out.println(stats);
14.	// DoubleSummaryStatistics{count=4, sum=10.620000, min=1.100000, average=2.655000, max=4.040000}

Все эти методы и несколько последующих, зачастую используются в качестве составных коллекторов для группировки или collectingAndThen. В том виде, в котором они показаны на примерах используются редко. Я лишь показываю пример, что они возвращают, чтобы было

понятнее.

counting()

Подсчитывает количество элементов.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Long count = Stream.of("1", "2", "3", "4")
2.   .collect(Collectors.counting());
3. System.out.println(count);
4. // 4
```

filtering(Predicate predicate, Collector downstream)

mapping(Function mapper, Collector downstream)

flatMapMapping(Function downstream)

reducing(BinaryOperator op)

reducing(T identity, BinaryOperator op)

reducing(U identity, Function mapper, BinaryOperator op)

Специальная группа коллекторов, которая применяет операции filter, map, flatMap и reduce.

filtering и flatMapping появились в Java 9.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. List<Integer> ints = Stream.of(1, 2, 3, 4, 5, 6)
2.   .collect(Collectors.filtering(
3.       x -> x % 2 == 0,
4.       Collectors.toList()));
5. // 2, 4, 6
6.
7. String s1 = Stream.of(1, 2, 3, 4, 5, 6)
8.   .collect(Collectors.filtering(
9.       x -> x % 2 == 0,
10.      Collectors.mapping(
11.          x -> Integer.toString(x),
12.          Collectors.joining("-"))
13.      )));
14. // 2-4-6
15.
16.
17. String s2 = Stream.of(2, 0, 1, 3, 2)
18.   .collect(Collectors.flatMapMapping(
19.       x -> IntStream.range(0, x).mapToObj(Integer::toString),
20.       Collectors.joining(", "))
21.   ));
22. // 0, 1, 0, 0, 1, 2, 0, 1
23.
24. int value = Stream.of(1, 2, 3, 4, 5, 6)
25.   .collect(Collectors.reducing(
26.       0, (a, b) -> a + b
27.   ));
28. // 21
29. String s3 = Stream.of(1, 2, 3, 4, 5, 6)
30.   .collect(Collectors.reducing(
31.       "", x -> Integer.toString(x), (a, b) -> a + b
32.   ));
33. // 123456
```

minBy(Comparator comparator)

maxBy(Comparator comparator)

Поиск минимального/максимального элемента, основываясь на заданном компараторе.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Optional<String> min = Stream.of("ab", "c", "defgh", "ijk", "l")
2.   .collect(Collectors.minBy(Comparator.comparing(String::length)));
3. min.ifPresent(System.out::println);
4. // c
5.
6. Optional<String> max = Stream.of("ab", "c", "defgh", "ijk", "l")
7.   .collect(Collectors.maxBy(Comparator.comparing(String::length)));
8. max.ifPresent(System.out::println);
9. // defgh
```


groupingBy(Function classifier)
groupingBy(Function classifier, Collector downstream)
groupingBy(Function classifier, Supplier mapFactory, Collector downstream)

Группирует элементы по критерию, сохраняя результат в Map. Вместе с представленными выше агрегирующими коллекторами, позволяет гибко собирать данные. Подробнее о комбинировании в разделе [Примеры \(#examples\)](#).

groupingByConcurrent(Function classifier)
groupingByConcurrent(Function classifier, Collector downstream)
groupingByConcurrent(Function classifier, Supplier mapFactory, Collector downstream)

Аналогичный набор методов, только сохраняет в ConcurrentMap.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1.  Map<Integer, List<String>> map1 = Stream.of(  
2.      "ab", "c", "def", "gh", "ijk", "l", "mnop")  
3.      .collect(Collectors.groupingBy(String::length));  
4.  map1.entrySet().forEach(System.out::println);  
5.  // 1=[c, l]  
6.  // 2=[ab, gh]  
7.  // 3=[def, ijk]  
8.  // 4=[mnop]  
9.  
10. Map<Integer, String> map2 = Stream.of(  
11.     "ab", "c", "def", "gh", "ijk", "l", "mnop")  
12.     .collect(Collectors.groupingBy(  
13.         String::length,  
14.         Collectors.mapping(  
15.             String::toUpperCase,  
16.             Collectors.joining())  
17.     ));  
18. map2.entrySet().forEach(System.out::println);  
19. // 1=CL  
20. // 2=ABGH  
21. // 3=DEFIJK  
22. // 4=MNOP  
23.  
24. Map<Integer, List<String>> map3 = Stream.of(  
25.     "ab", "c", "def", "gh", "ijk", "l", "mnop")  
26.     .collect(Collectors.groupingBy(  
27.         String::length,  
28.         LinkedHashMap::new,  
29.         Collectors.mapping(  
30.             String::toUpperCase,  
31.             Collectors.toList())  
32.     ));  
33. map3.entrySet().forEach(System.out::println);  
34. // 2=[AB, GH]  
35. // 1=[C, L]  
36. // 3=[DEF, IJK]  
37. // 4=[MNOP]
```

partitioningBy(Predicate predicate)
partitioningBy(Predicate predicate, Collector downstream)

Ещё один интересный метод. Разбивает последовательность элементов по какому-либо критерию. В одну часть попадают все элементы, которые удовлетворяют переданному условию, во вторую — все, которые не удовлетворяют.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1.  Map<Boolean, List<String>> map1 = Stream.of(  
2.      "ab", "c", "def", "gh", "ijk", "l", "mnop")  
3.      .collect(Collectors.partitioningBy(s -> s.length() <= 2));  
4.  map1.entrySet().forEach(System.out::println);  
5.  // false=[def, ijk, mnop]  
6.  // true=[ab, c, gh, l]  
7.  
8.  Map<Boolean, String> map2 = Stream.of(  
9.      "ab", "c", "def", "gh", "ijk", "l", "mnop")  
10.     .collect(Collectors.partitioningBy(  
11.         s -> s.length() <= 2,  
12.         Collectors.mapping(  
13.             String::toUpperCase,  
14.             Collectors.joining())  
15.     ));  
16. map2.entrySet().forEach(System.out::println);
```

17.	// false=DEFIJKMNOP
18.	// true=ABCGHL

8. Collector

Интерфейс `java.util.stream.Collector` служит для сбора элементов стрима в некоторый мутабельный контейнер. Он состоит из таких методов:

- `Supplier<A> supplier()` — функция, которая создаёт экземпляры контейнеров.
- `BiConsumer<A,T> accumulator()` — функция, которая кладёт новый элемент в контейнер.
- `BinaryOperator<A> combiner()` — функция, которая объединяет два контейнера в один. В параллельных стримах каждая часть может собираться в отдельный экземпляр контейнера и в итоге необходимо их объединять в один результирующий.
- `Function<A,R> finisher()` — функция, которая преобразовывает весь контейнер в конечный результат. Например, можно обернуть `List` в `Collections.unmodifiableList`.
- `Set<Characteristics> characteristics()` — возвращает характеристики коллেকтора, чтобы внутренняя реализация знала, с чем имеет дело. Например, можно указать, что коллектор поддерживает многопоточность.

Характеристики:

- `CONCURRENT` — коллектор поддерживает многопоточность, а значит отдельные части стрима могут быть успешно положены в контейнер из другого потока.
- `UNORDERED` — коллектор не зависит от порядка поступаемых элементов.
- `IDENTITY_FINISH` — функция `finish()` имеет стандартную реализацию (`Function.identity()`), а значит её можно не вызывать.

8.1. Реализация собственного коллектора

Прежде чем писать свой коллектор, нужно убедиться, что задачу нельзя решить при помощи комбинации стандартных коллекторов. К примеру, если нужно собрать лишь уникальные элементы в список, то можно собрать элементы сначала в `LinkedHashSet`, чтобы сохранился порядок, а потом все элементы добавить в `ArrayList`. Комбинация `collectingAndThen` с `toCollection` и функцией, передающей полученный `Set` в конструктор `ArrayList`, делает то, что задумано:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>Stream.of(1, 2, 3, 1, 9, 2, 5, 3, 4, 8, 2)</code>
2.	<code>.collect(Collectors.collectingAndThen(</code>
3.	<code>Collectors.toCollection(LinkedHashSet::new),</code>
4.	<code>ArrayList::new));</code>
5.	<code>// 1 2 3 9 5 4 8</code>

А вот если задача состоит в том, чтобы собрать уникальные элементы в одну часть, а повторяющиеся в другую, например в `Map<Boolean, List>`, то при помощи `partitioningBy` получится не очень красиво:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>final Set<Integer> elements = new HashSet<>();</code>
2.	<code>Stream.of(1, 2, 3, 1, 9, 2, 5, 3, 4, 8, 2)</code>
3.	<code>.collect(Collectors.partitioningBy(elements::add))</code>
4.	<code>.forEach((isUnique, list) -> System.out.format("%s: %s\n", isUnique ? "unique" : "repetitive", list));</code>

Здесь приходится создавать `Set` и в предикате коллектора его использовать, что нежелательно. Можно превратить лямбду в анонимную функцию, но это ещё хуже:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>new Predicate<Integer>() {</code>
2.	<code>final Set<Integer> elements = new HashSet<>();</code>
3.	<code>@Override</code>
4.	<code>public boolean test(Integer t) {</code>
5.	<code>return elements.add(t);</code>
6.	<code>}</code>
7.	<code>}</code>

Для создания своего коллектора есть два пути:

1. Создать класс, реализующий интерфейс `Collector`.
2. Воспользоваться фабрикой `Collector.of`.

Если нужно сделать что-то универсальное, чтобы работало для любых типов, то есть использовать дженерики, то во втором варианте можно просто сделать статическую функцию, а внутри использовать `Collector.of`.

Вот полученный коллектор.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. public static <T> Collector<T, ?, Map<Boolean, List<T>>> partitioningByUniqueness()
2. {
3.     return Collector.<T, Map.Entry<List<T>, Set<T>>, Map<Boolean, List<T>>>of(
4.         () -> new AbstractMap.SimpleImmutableEntry<>(
5.             new ArrayList<T>(), new LinkedHashSet<>()),
6.         (c, e) -> {
7.             if (!c.getValue().add(e)) {
8.                 c.getKey().add(e);
9.             }
10.        },
11.        (c1, c2) -> {
12.            c1.getKey().addAll(c2.getKey());
13.            for (T e : c2.getValue()) {
14.                if (!c1.getValue().add(e)) {
15.                    c1.getKey().add(e);
16.                }
17.            }
18.            return c1;
19.        },
20.        c -> {
21.            Map<Boolean, List<T>> result = new HashMap<>(2);
22.            result.put(Boolean.FALSE, c.getKey());
23.            result.put(Boolean.TRUE, new ArrayList<>(c.getValue()));
24.            return result;
25.        });
26. }
```

Давайте теперь разбираться.

Интерфейс Collector объявлен так:

```
interface Collector<T, A, R>
    T - тип входных элементов.
    A - тип контейнера, в который будут поступать элементы.
    R - тип результата.
```

Сигнатура метода, возвращающего коллектор такова:

```
public static <T> Collector<T, ?, Map<Boolean, List<T>>>
partitioningByUniqueness()
```

Он принимает элементы типа T, возвращает Map<Boolean, List<T>>, как и partitioningBy. Знак вопроса (джокер) в среднем параметре говорит о том, что внутренний тип реализации для публичного API не важен. Многие методы класса Collectors содержат джокер в качестве типа контейнера.

```
return Collector.<T, Map.Entry<List<T>, Set<T>>, Map<Boolean, List<T>>>of
Вот здесь уже пришлось указать тип контейнера. Так как в Java нет класса Pair или Tuple, то два разных типа можно положить в Map.Entry.
```

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. // supplier
2. () -> new AbstractMap.SimpleImmutableEntry<>(
3.     new ArrayList<>(), new LinkedHashSet<>())
```

Контейнером будет AbstractMap.SimpleImmutableEntry. В ключе будет список повторяющихся элементов, в значении — множество с уникальными элементами.

```
1. // accumulator
2. (c, e) -> {
3.     if (!c.getValue().add(e)) {
4.         c.getKey().add(e);
5.     }
6. }
```

Здесь всё просто. Если элемент нельзя добавить во множество (по причине того, что там уже есть такой элемент), то добавляем его в список повторяющихся элементов.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. // combiner
2. (c1, c2) -> {
3.     c1.getKey().addAll(c2.getKey());
4.     for (T e : c2.getValue()) {
5.         if (!c1.getValue().add(e)) {
```

```
6.         c1.getKey().add(e);
7.     }
8. }
9. return c1;
10. }
```

Нужно объединить два Map.Entry. Списки повторяющихся элементов можно объединить вместе, а вот с уникальными элементами так просто не выйдет — нужно пройтись поэлементно и повторить всё то, что делалось в функции-аккумуляторе. Кстати, лямбду-аккумулятор можно присвоить переменной и тогда цикл можно превратить в

```
c2.getValue().forEach(e -> accumulator.accept(c1, e));
```

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. // finisher
2. c -> {
3.     Map<Boolean, List<T>> result = new HashMap<>(2);
4.     result.put(Boolean.FALSE, c.getKey());
5.     result.put(Boolean.TRUE, new ArrayList<>(c.getValue()));
6.     return result;
7. }
```

Наконец, возвращаем необходимый результат. В map.get(Boolean.TRUE) будут уникальные, а в map.get(Boolean.FALSE) — повторяющиеся элементы.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. Map<Boolean, List<Integer>> map;
2. map = Stream.of(1, 2, 3, 1, 9, 2, 5, 3, 4, 8, 2)
3.     .collect(partitioningByUniqueness());
4. // {false=[1, 2, 3, 2], true=[1, 2, 3, 9, 5, 4, 8]}
```

Хорошей практикой является создание коллекторов, которые принимают ещё один коллектор и зависят от него. Например, можно будет складывать элементы не только в List, но и в любую другую коллекцию (Collectors.toCollection), либо в строку (Collectors.joining).

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. public static <T, D, A> Collector<T, ?, Map<Boolean, D>> partitioningByUniqueness(
2.     Collector<? super T, A, D> downstream) {
3.     class Holder<A, B> {
4.         final A unique, repetitive;
5.         final B set;
6.         Holder(A unique, A repetitive, B set) {
7.             this.unique = unique;
8.             this.repetitive = repetitive;
9.             this.set = set;
10.        }
11.    }
12.    BiConsumer<A, ? super T> downstreamAccumulator = downstream.accumulator();
13.    BinaryOperator<A> downstreamCombiner = downstream.combiner();
14.    BiConsumer<Holder<A, Set<T>>, T> accumulator = (t, element) -> {
15.        A container = t.set.add(element) ? t.unique : t.repetitive;
16.        downstreamAccumulator.accept(container, element);
17.    };
18.    return Collector.<T, Holder<A, Set<T>>, Map<Boolean, D>>>of(
19.        () -> new Holder<>(
20.            downstream.supplier().get(),
21.            downstream.supplier().get(),
22.            new HashSet<>() ),
23.        accumulator,
24.        (t1, t2) -> {
25.            downstreamCombiner.apply(t1.repetitive, t2.repetitive);
26.            t2.set.forEach(e -> accumulator.accept(t1, e));
27.            return t1;
28.        },
29.        t -> {
30.            Map<Boolean, D> result = new HashMap<>(2);
31.            result.put(Boolean.FALSE,
32.                downstream.finisher().apply(t.repetitive));
33.            result.put(Boolean.TRUE, downstream.finisher().apply(t.unique));
34.            t.set.clear();
35.            return result;
36.        });
36. }
```

Алгоритм остался тем же, только теперь уже нельзя во второй контейнер сразу же складывать уникальные элементы, приходится создавать новый set. Для удобства также добавлен класс Holder, который хранит два контейнера для уникальных и повторяющихся элементов, а также само множество.

Все операции теперь нужно проводить через переданный коллектор, именуемый downstream. Именно он сможет поставить контейнер нужного типа (downstream.supplier().get()), добавить элемент в этот контейнер (downstream.accumulator().accept(container, element)), объединить контейнеры и создать окончательный результат.

```
[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)
1. Stream.of(1, 2, 3, 1, 9, 2, 5, 3, 4, 8, 2)
2.     .map(String::valueOf)
3.     .collect(partitioningByUniqueness(Collectors.joining("-")))
4.     .forEach((isUnique, str) -> System.out.format("%s: %s%n", isUnique ? "unique" :
"repetitive", str));
5. // repetitive: 1-2-3-2
6. // unique: 1-2-3-9-5-4-8
```

Кстати, первую реализацию метода без аргументов можно теперь заменить на:

```
[скопировать] (javascript:void(0);) [скачать] (javascript:void(0);)
1. public static <T> Collector<T, ?, Map<Boolean, List<T>>> partitioningByUniqueness()
   {
2.     return partitioningByUniqueness(Collectors.toList());
3. }
```

9. Spliterator

Пришло время немного углубиться в работу Stream API изнутри. Элементы стримов нужно не только итерировать, но ещё и разделять на части и отдавать другим потокам. За итерацию и разбиение отвечает Spliterator. Он даже звучит как Iterator, только с приставкой Split — разделять.

Методы интерфейса:

- trySplit — как следует из названия, пытается разделить элементы на две части. Если это сделать не получается, либо элементов недостаточно для деления, то вернёт null. В остальных случаях возвращает ещё один Spliterator с частью данных.
- tryAdvance(Consumer action) — если имеются элементы, для которых можно применить действие, то оно применяется и возвращает true, в противном случае возвращается false, но действие не выполняется.
- estimateSize() — возвращает примерное количество элементов, оставшихся для обработки, либо Long.MAX_VALUE, если стрим бесконечный или посчитать количество невозможно.
- characteristics() — возвращает характеристики сплитератора.

9.1. Характеристики

В методе sorted и distinct было упомянуто, что если стрим помечен как отсортированный или содержащий уникальные элементы, то соответствующие операции проводиться не будут. Вот характеристики сплитератора и влияют на это.

- DISTINCT — все элементы уникальны. Сплитераторы всех реализаций Set содержат эту характеристику.
- SORTED — все элементы отсортированы.
- ORDERED — порядок имеет значение. Сплитераторы большинства коллекций содержат эту характеристику, а HashSet, к примеру, нет.
- SIZED — количество элементов точно известно.
- SUBSIZED — количество элементов каждой разбитой части точно известно.
- NONNULL — в элементах не встречается null. Некоторые коллекции из java.util.concurrent, в которые нельзя положить null, содержат эту характеристику.
- IMMUTABLE — источник является иммутабельным и в него нельзя больше добавить элементов, либо удалить их.
- CONCURRENT — источник лоялен к любым изменениям.

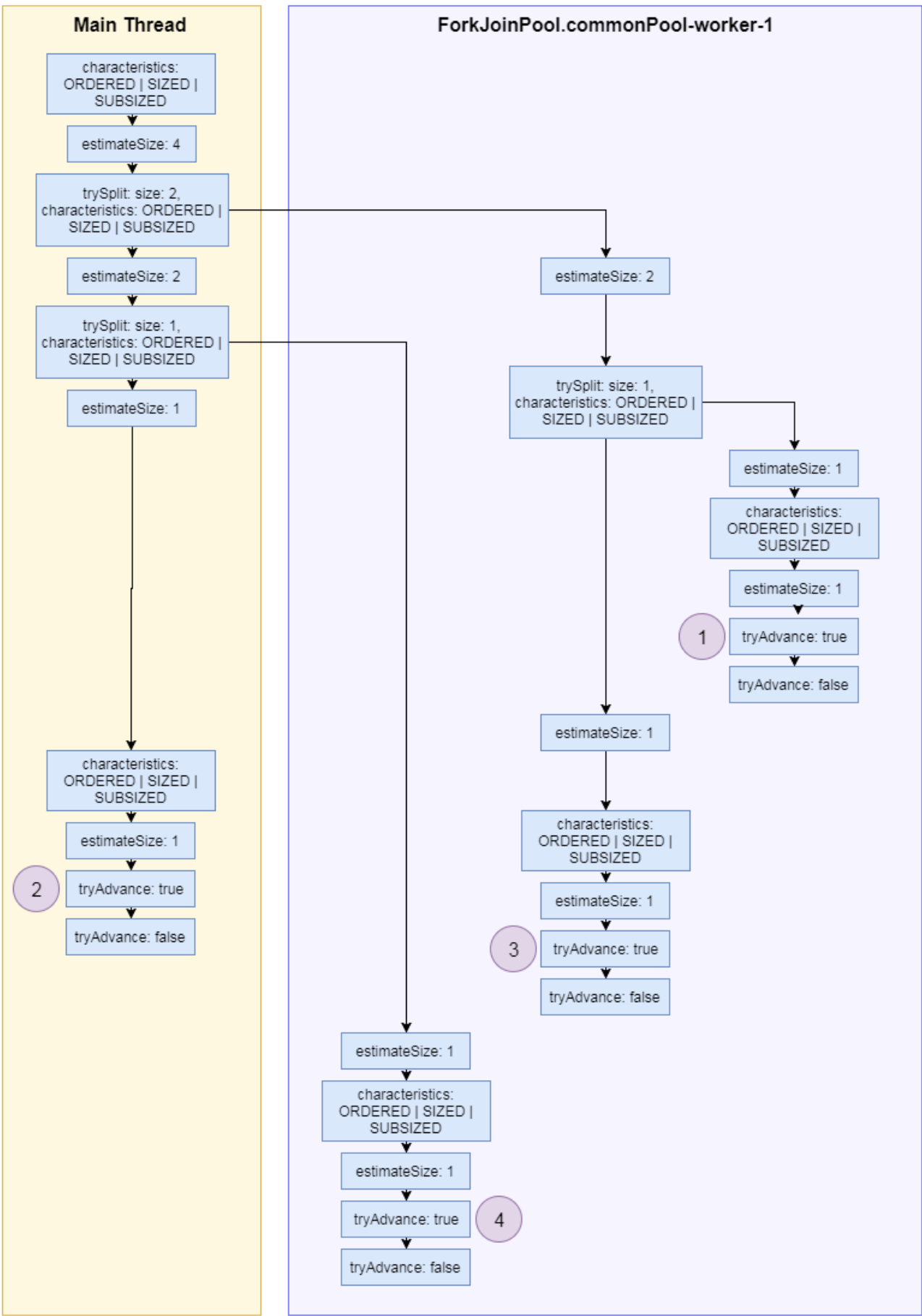
Разумеется, характеристики могут быть изменены при выполнении цепочки операторов. Например, после sorted добавляется характеристика SORTED, после filter теряется SIZED и т.д.

9.2. Жизненный цикл сплитератора

Чтобы понять когда и как сплитератор вызывает тот или иной метод, давайте создадим обёртку, которая [логирует все вызовы](#) (<https://gist.github.com/aNNiMON/71488fe8bfc6781d641fd4d0ed1f1aa7>). Чтобы из сплитератора

создать стрим, используется класс StreamSupport.

```
1. long count = StreamSupport.stream(
2.     Arrays.asList(0, 1, 2, 3).spliterator(), true)
3.     .count();
```



На рисунке показан один из возможных вариантов работы сплитератора. characteristics везде возвращает ORDERED | SIZED | SUBSIZED, так как в List порядок имеет значение, количество элементов и всех разбитых кусков также известно. trySplit делит последовательность пополам, но не обязательно каждая часть будет отправлена новому потоку. В параллельном стриме новый поток может и не создаваться, т.к. всё успеваает обработаться в главном потоке. В данном же случае, новый поток успевал обработать части до того, как это делал главный поток.

```
[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)
1. Spliterator<Integer> s = IntStream.range(0, 4)
2.     .boxed()
3.     .collect(Collectors.toSet())
4.     .spliterator();
5. long count = StreamSupport.stream(s, true).count();
```

Здесь у сплитератора характеристикой будет SIZED | DISTINCT, а вот у каждой части характеристика SIZED теряется, остаётся только DISTINCT, потому что нельзя поделить множество так, чтобы размер каждой части был известен.

В случае с Set было три вызова trySplit, первый якобы делил элементы поровну, после двух других каждая из частей возвращала estimateSize: 1, однако во всех, кроме одной попытка вызвать tryAdvance не увенчалась успехом — возвращался false. А вот на одном из частей, который для estimateSize также возвращал 1, было 4 успешных вызова tryAdvance. Это и подтверждает тот факт, что estimateSize не обязательно должен возвращать действительное

число элементов.

1.	Arrays.spliterator(new int[] {0, 1, 2, 3});
2.	Stream.of(0, 1, 2, 3).spliterator();

Ситуация аналогична работе List, только характеристики возвращали ORDERED | SIZED | SUBSIZED | IMMUTABLE.

1.	Stream.of(0, 1, 2, 3).distinct().spliterator();
----	-------------------------------------------------

Здесь trySplit возвращал null, а значит поделить последовательно не представлялось возможным. Иерархия вызовов:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	[main] characteristics: ORDERED DISTINCT
2.	[main] estimateSize: 4
3.	[main] trySplit: null
4.	[main] characteristics: ORDERED DISTINCT
5.	[main] tryAdvance: true
6.	[main] tryAdvance: true
7.	[main] tryAdvance: true
8.	[main] tryAdvance: true
9.	[main] tryAdvance: false
10.	count: 4

1.	Stream.of(0, 1, 2, 3)
2.	.distinct()
3.	.map(x -> x + 1)
4.	.spliterator();

Всё, как и выше, только теперь после применения оператора map, флаг DISTINCT исчез.

9.3. Реализация сплитератора

Для правильной реализации сплитератора нужно продумать, как сделать разбиение и обозначить характеристики стрима. Давайте напишем сплитератор, генерирующий последовательность чисел Фибоначчи.

Для упрощения задачи нам будет известно максимальное количество элементов для генерирования. А значит мы можем разделять последовательность пополам, а потом быстро просчитывать нужные числа по новому индексу.

Осталось определиться с характеристиками. Мы уже решили, что размер последовательности нам будет известен, а значит будет известен и размер каждой разбитой части. Порядок будет важен, так что без флага ORDERED не обойтись. Последовательность Фибоначчи также отсортирована — каждый последующий элемент всегда не меньше предыдущего.

А вот с флагом DISTINCT, кажется, промах. 0 1 1 2 3, две единицы повторяются, а значит не видать нам этого флага?

На самом деле ничто нам не мешает просчитывать флаги автоматически. Если часть последовательности не будет затрагивать начальные индексы, то этот флаг можно выставить.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	int distinct = (index >= 2) ? DISTINCT : 0;
2.	return ORDERED distinct SIZED SUBSIZED IMMUTABLE NONNULL;

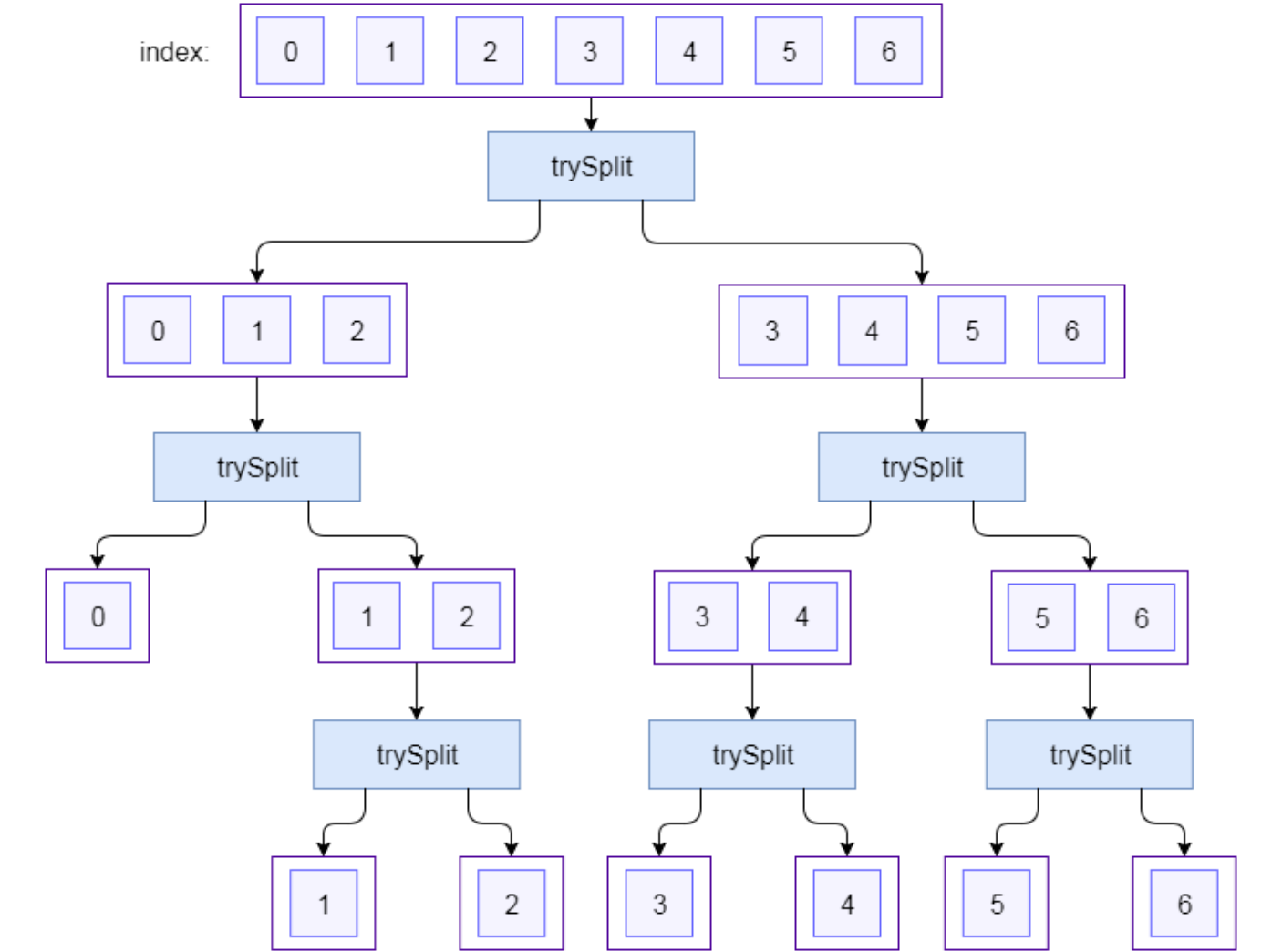
Полная реализация класса:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	import java.math.BigInteger;
2.	import java.util.Spliterator;
3.	import java.util.function.Consumer;
4.	
5.	public class FibonacciSpliterator implements Spliterator<BigInteger> {
6.	
7.	private final int fence;
8.	private int index;
9.	private BigInteger a, b;
10.	
11.	public FibonacciSpliterator(int fence) {
12.	this(0, fence);
13.	}
14.	
15.	protected FibonacciSpliterator(int start, int fence) {
16.	this.index = start;
17.	this.fence = fence;

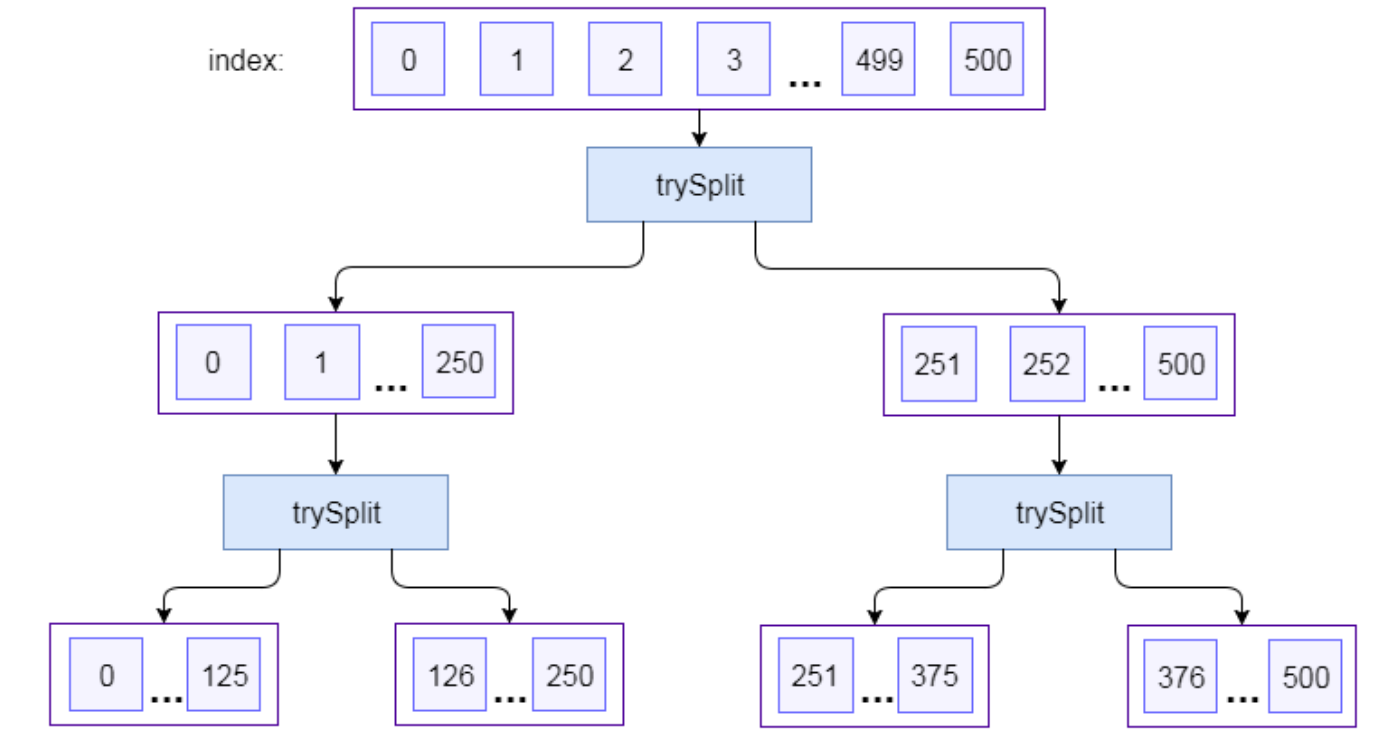

```
18.         recalculateNumbers(start);
19.     }
20.
21.     private void recalculateNumbers(int start) {
22.         a = fastFibonacciDoubling(start);
23.         b = fastFibonacciDoubling(start + 1);
24.     }
25.
26.     @Override
27.     public boolean tryAdvance(Consumer<? super BigInteger> action) {
28.         if (index >= fence) {
29.             return false;
30.         }
31.         action.accept(a);
32.         BigInteger c = a.add(b);
33.         a = b;
34.         b = c;
35.         index++;
36.         return true;
37.     }
38.
39.     @Override
40.     public FibonacciSpliterator trySplit() {
41.         int lo = index;
42.         int mid = (lo + fence) >>> 1;
43.         if (lo >= mid) {
44.             return null;
45.         }
46.         index = mid;
47.         recalculateNumbers(mid);
48.         return new FibonacciSpliterator(lo, mid);
49.     }
50.
51.     @Override
52.     public long estimateSize() {
53.         return fence - index;
54.     }
55.
56.     @Override
57.     public int characteristics() {
58.         int distinct = (index >= 2) ? DISTINCT : 0;
59.         return ORDERED | distinct | SIZED | SUBSIZED | IMMUTABLE | NONNULL;
60.     }
61.
62.     /*
63.      * https://www.nayuki.io/page/fast-fibonacci-algorithms
64.      */
65.     public static BigInteger fastFibonacciDoubling(int n) {
66.         BigInteger a = BigInteger.ZERO;
67.         BigInteger b = BigInteger.ONE;
68.         for (int bit = Integer.highestOneBit(n); bit != 0; bit >>= 1) {
69.             BigInteger d = a.multiply(b.shiftLeft(1).subtract(a));
70.             BigInteger e = a.multiply(a).add(b.multiply(b));
71.             a = d;
72.             b = e;
73.             if ((n & bit) != 0) {
74.                 BigInteger c = a.add(b);
75.                 a = b;
76.                 b = c;
77.             }
78.         }
79.         return a;
80.     }
81. }
```

Вот как разбиваются теперь элементы параллельного стрима:

1.	StreamSupport.stream(new FibonacciSpliterator(7), true)
2.	.count();



```
1. StreamSupport.stream(new FibonacciSpliterator(500), true)
2. .count();
```



10. Другие способы создания источников

Стрим из сплитератора — это самый эффективный способ создания стрима, но кроме него есть и другие способы.

10.1. Стрим из итератора

Благодаря классу Spliterators, можно преобразовать любой итератор в сплитератор. Вот пример создания стрима из итератора, генерирующего бесконечную последовательность чисел Фибоначчи.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. public class FibonacciIterator implements Iterator<BigInteger> {
2.
3.     private BigInteger a = BigInteger.ZERO;
4.     private BigInteger b = BigInteger.ONE;
5.
6.     @Override
7.     public boolean hasNext() {
8.         return true;
9.     }
10.
11.     @Override
```

12.	<code>public BigInteger next() {</code>
13.	<code> BigInteger result = a;</code>
14.	<code> a = b;</code>
15.	<code> b = result.add(b);</code>
16.	<code> return result;</code>
17.	<code>}</code>
18.	<code>}</code>
19.	
20.	<code>StreamSupport.stream(</code>
21.	<code> Spliterators.spliteratorUnknownSize(</code>
22.	<code> new FibonacciIterator(),</code>
23.	<code> Spliterator.ORDERED Spliterator.SORTED),</code>
24.	<code>false /* is parallel */)</code>
25.	<code>.limit(10)</code>
26.	<code>.forEach(System.out::println);</code>

10.2. Stream.iterate + map

Можно воспользоваться двумя операторами: `iterate` + `map`, чтобы создать всё тот же стрим из чисел Фибоначчи.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>Stream.iterate(</code>
2.	<code> new BigInteger[] { BigInteger.ZERO, BigInteger.ONE },</code>
3.	<code> t -> new BigInteger[] { t[1], t[0].add(t[1]) }</code>
4.	<code>.map(t -> t[0])</code>
5.	<code>.limit(10)</code>
6.	<code>.forEach(System.out::println);</code>

Для удобства можно обернуть всё в метод и вызывать `fibonacciStream().limit(10).forEach(...)`.

10.3. IntStream.range + map

Ещё один гибкий и удобный способ создать стрим. Если у вас есть какие-то данные, которые можно получить по индексу, то можно создать числовой промежуток при помощи оператора `range`, затем поэлементно с помощью него обращаться к данным через `map/mapToObj`.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>IntStream.range(0, 200)</code>
2.	<code>.mapToObj(i -> fibonacci(i))</code>
3.	<code>.forEach(System.out::println);</code>
4.	
5.	<code>JSONArray arr = ...</code>
6.	<code>IntStream.range(0, arr.length())</code>
7.	<code>.mapToObj(JSONArray::getJSONObject)</code>
8.	<code>.map(obj -> ...)</code>
9.	<code>.forEach(System.out::println);</code>

11. Примеры

Прежде чем перейти к более приближенным к жизни примерам, стоит сказать, что если код уже написан без стримов и работает хорошо, не нужно сломя голову всё переписывать. Также бывает ситуации, когда красиво реализовать задачу с использованием `Stream API` не получается, в таком случае смириться и не тяните стримы за уши.

Дан массив аргументов. Нужно получить `Map`, где каждому ключу будет соответствовать своё значение.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)	
1.	<code>String[] arguments = {"-i", "in.txt", "--limit", "40", "-d", "1", "-o", "out.txt"};</code>
2.	<code>Map<String, String> argsMap = new LinkedHashMap<>(arguments.length / 2);</code>
3.	<code>for (int i = 0; i < arguments.length; i += 2) {</code>
4.	<code> argsMap.put(arguments[i], arguments[i + 1]);</code>
5.	<code>}</code>
6.	<code>argsMap.forEach((key, value) -> System.out.format("%s: %s%n", key, value));</code>
7.	<code>// -i: in.txt</code>
8.	<code>// --limit: 40</code>
9.	<code>// -d: 1</code>
10.	<code>// -o: out.txt</code>

Быстро и понятно. А вот для обратной задачи — сконвертировать `Map` с аргументами в массив строк, стримы помогут.

[копировать] (javascript:void(0)); [скачать] (javascript:void(0));

```
1. String[] args = argsMap.entrySet().stream()
2.     .flatMap(e -> Stream.of(e.getKey(), e.getValue()))
3.     .toArray(String[]::new);
4. System.out.println(String.join(" ", args));
5. // -i in.txt --limit 40 -d 1 -o out.txt
```

Дан список студентов.

[копировать] (javascript:void(0)); [скачать] (javascript:void(0));

```
1. List<Student> students = Arrays.asList(
2.     new Student("Alex", Speciality.Physics, 1),
3.     new Student("Rika", Speciality.Biology, 4),
4.     new Student("Julia", Speciality.Biology, 2),
5.     new Student("Steve", Speciality.History, 4),
6.     new Student("Mike", Speciality.Finance, 1),
7.     new Student("Hinata", Speciality.Biology, 2),
8.     new Student("Richard", Speciality.History, 1),
9.     new Student("Kate", Speciality.Psychology, 2),
10.    new Student("Sergey", Speciality.ComputerScience, 4),
11.    new Student("Maximilian", Speciality.ComputerScience, 3),
12.    new Student("Tim", Speciality.ComputerScience, 5),
13.    new Student("Ann", Speciality.Psychology, 1)
14. );
15.
16. enum Speciality {
17.     Biology, ComputerScience, Economics, Finance,
18.     History, Philosophy, Physics, Psychology
19. }
```

У класса Student реализованы все геттеры и сеттеры, toString и equals+hashCode.

Нужно сгруппировать всех студентов по курсу.

[копировать] (javascript:void(0)); [скачать] (javascript:void(0));

```
1. students.stream()
2.     .collect(Collectors.groupingBy(Student::getYear))
3.     .entrySet().forEach(System.out::println);
4. // 1=[Alex: Physics 1, Mike: Finance 1, Richard: History 1, Ann: Psychology 1]
5. // 2=[Julia: Biology 2, Hinata: Biology 2, Kate: Psychology 2]
6. // 3=[Maximilian: ComputerScience 3]
7. // 4=[Rika: Biology 4, Steve: History 4, Sergey: ComputerScience 4]
8. // 5=[Tim: ComputerScience 5]
```

Вывести в алфавитном порядке список специальностей, на которых учатся перечисленные в списке студенты.

[копировать] (javascript:void(0)); [скачать] (javascript:void(0));

```
1. students.stream()
2.     .map(Student::getSpeciality)
3.     .distinct()
4.     .sorted(Comparator.comparing(Enum::name))
5.     .forEach(System.out::println);
6. // Biology
7. // ComputerScience
8. // Finance
9. // History
10. // Physics
11. // Psychology
```

Вывести количество учащихся на каждой из специальностей.

[копировать] (javascript:void(0)); [скачать] (javascript:void(0));

```
1. students.stream()
2.     .collect(Collectors.groupingBy(
3.         Student::getSpeciality, Collectors.counting()))
4.     .forEach((s, count) -> System.out.println(s + ": " + count));
5. // Psychology: 2
6. // Physics: 1
7. // ComputerScience: 3
8. // Finance: 1
9. // Biology: 3
10. // History: 2
```

Сгруппировать студентов по специальностям, сохраняя алфавитный порядок специальности, а затем сгруппировать по курсу.

[копировать] (javascript:void(0)); [скачать] (javascript:void(0));

```
1. Map<Speciality, Map<Integer, List<Student>>> result = students.stream()
2.     .sorted(Comparator
3.         .comparing(Student::getSpeciality, Comparator.comparing(Enum::name))
4.         .thenComparing(Student::getYear)
5.     )
6.     .collect(Collectors.groupingBy(
7.         Student::getSpeciality,
8.         LinkedHashMap::new,
9.         Collectors.groupingBy(Student::getYear)));
```

Теперь это всё красиво вывести.

```
[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)
1. result.forEach((s, map) -> {
2.     System.out.println("-= " + s + " -=");
3.     map.forEach((year, list) -> System.out.format("%d: %s%n", year, list.stream()
4.         .map(Student::getName)
5.         .sorted()
6.         .collect(Collectors.joining(", "))))
7. });
8. System.out.println();
9. });
```

-- Biology --
2: Hinata, Julia
4: Rika

-- ComputerScience --
3: Maximilian
4: Sergey
5: Tim

-- Finance --
1: Mike

-- History --
1: Richard
4: Steve

-- Physics --
1: Alex

-- Psychology --
1: Ann
2: Kate

Проверить, есть ли третьекурсники среди учащихся всех специальностей кроме физики и CS.

```
[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)
1. students.stream()
2.     .filter(s -> !EnumSet.of(Speciality.ComputerScience, Speciality.Physics)
3.         .contains(s.getSpeciality()))
4.     .anyMatch(s -> s.getYear() == 3);
5. // false
```

Вычислить число Пи методом Монте-Карло.

```
[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)
1. final Random rnd = new Random();
2. final double r = 1000.0;
3. final int max = 10000000;
4. long count = IntStream.range(0, max)
5.     .mapToObj(i -> rnd.doubles(2).map(x -> x * r).toArray())
6.     .parallel()
7.     .filter(arr -> Math.hypot(arr[0], arr[1]) <= r)
8.     .count();
9. System.out.println(4.0 * count / max);
10. // 3.1415344
```

Вывести таблицу умножения.

```
[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)
1. IntStream.rangeClosed(2, 9)
2.     .boxed()
```

```
3.         .flatMap(i -> IntStream.rangeClosed(2, 9))
4.         .mapToObj(j -> String.format("%d * %d = %d", i, j, i * j))
5.         )
6.         .forEach(System.out::println);
7. // 2 * 2 = 4
8. // 2 * 3 = 6
9. // 2 * 4 = 8
10. // 2 * 5 = 10
11. // ...
12. // 9 * 7 = 63
13. // 9 * 8 = 72
14. // 9 * 9 = 81
```

Или более экзотический вариант, в 4 столбца, как на школьных тетрадах.

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1.  IntFunction<IntFunction<String>>> function = i -> j -> String.format("%d x %2d =
2.  IntFunction<IntFunction<IntFunction<String>>>> repeaterX = count -> i -> j ->
3.      IntStream.range(0, count)
4.          .mapToObj(delta -> function.apply(i + delta).apply(j))
5.          .collect(Collectors.joining("\t"));
6.  IntFunction<IntFunction<IntFunction<IntFunction<String>>>>> repeaterY = countY ->
7.      IntStream.range(0, countY)
8.          .mapToObj(deltaY -> repeaterX.apply(countX).apply(i).apply(j +
9.      deltaY))
10.         .collect(Collectors.joining("\n"));
11. IntFunction<String> row = i -> repeaterY.apply(10).apply(4).apply(i).apply(1) +
    "\n";
11. IntStream.of(2, 6).mapToObj(row).forEach(System.out::println);
```

ТАБЛИЦА УМНОЖЕНИЯ															
2 x 1 = 2	3 x 1 = 3	4 x 1 = 4	5 x 1 = 5	6 x 1 = 6	7 x 1 = 7	8 x 1 = 8	9 x 1 = 9	2 x 2 = 4	3 x 2 = 6	4 x 2 = 8	5 x 2 = 10	6 x 2 = 12	7 x 2 = 14	8 x 2 = 16	9 x 2 = 18
2 x 3 = 6	3 x 3 = 9	4 x 3 = 12	5 x 3 = 15	6 x 3 = 18	7 x 3 = 21	8 x 3 = 24	9 x 3 = 27	2 x 4 = 8	3 x 4 = 12	4 x 4 = 16	5 x 4 = 20	6 x 4 = 24	7 x 4 = 28	8 x 4 = 32	9 x 4 = 36
2 x 5 = 10	3 x 5 = 15	4 x 5 = 20	5 x 5 = 25	6 x 5 = 30	7 x 5 = 35	8 x 5 = 40	9 x 5 = 45	2 x 6 = 12	3 x 6 = 18	4 x 6 = 24	5 x 6 = 30	6 x 6 = 36	7 x 6 = 42	8 x 6 = 48	9 x 6 = 54
2 x 7 = 14	3 x 7 = 21	4 x 7 = 28	5 x 7 = 35	6 x 7 = 42	7 x 7 = 49	8 x 7 = 56	9 x 7 = 63	2 x 8 = 16	3 x 8 = 24	4 x 8 = 32	5 x 8 = 40	6 x 8 = 48	7 x 8 = 56	8 x 8 = 64	9 x 8 = 72
2 x 9 = 18	3 x 9 = 27	4 x 9 = 36	5 x 9 = 45	6 x 9 = 54	7 x 9 = 63	8 x 9 = 72	9 x 9 = 81	2 x 10 = 20	3 x 10 = 30	4 x 10 = 40	5 x 10 = 50	6 x 10 = 60	7 x 10 = 70	8 x 10 = 80	9 x 10 = 90

Но это, конечно же, шутка. Писать такой код вас никто не заставляет.

12. Задачи

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1.  IntStream.concat(
2.      IntStream.range(2, ____),
3.      IntStream.rangeClosed(____, ____))
4.      .forEach(System.out::println);
5. // 2, 3, 4, 5, -1, 0, 1, 2
6.
7. IntStream.range(5, 30)
8.     .limit(12)
9.     .skip(3)
10.    .limit(6)
11.    .skip(2)
12.    .forEach(System.out::println);
13. // ____, ____, ____, ____
14.
15. IntStream.range(0, 10)
16.     .skip(2)
17.     .dropWhile(x -> x < ____ )
18.     .limit(____)
19.     .forEach(System.out::println);
20. // 5, 6, 7
21.
22. IntStream.range(0, 10)
23.     .skip(____)
```



```
24.         .takeWhile(x -> x <           )
25.         .limit(3)
26.         .forEach(System.out::println);
27. // 3, 4
28.
29. IntStream.range(1, 5)
30.         .flatMap(i -> IntStream.generate(() ->           ).          (          ))
31.         .forEach(System.out::println);
32. // 1, 2, 2, 3, 3, 3, 4, 4, 4, 4
33.
34. int x = IntStream.range(-2, 2)
35.         .map(i -> i *           )
36.         .reduce(10, Integer::sum);
37. // x: 0
38.
39. IntStream.range(0, 10)
40.         .boxed()
41.         .collect(Collectors.                          (i ->                           ))
42.         .entrySet().forEach(System.out::println);
43. // false=[1, 3, 5, 7, 9]
44. // true=[0, 2, 4, 6, 8]
45.
46. IntStream.range(-5, 0)
47.         .flatMap(i -> IntStream.of(i,           ))
48.         .          ()
49.         .forEach(System.out::println);
50. // -5, -4, -3, -2, -1, 1, 2, 3, 4, 5
51.
52. IntStream.range(-5, 0)
53.         .flatMap(i -> IntStream.of(i,           ))
54.         .          ()
55.         .sorted(Comparator.comparing(Math::          ))
56.         .forEach(System.out::println);
57. // -1, 1, -2, 2, -3, 3, -4, 4, -5, 5
58.
59. IntStream.range(1, 5)
60.         .flatMap(i -> IntStream.generate(() -> i).limit(i))
61.         .boxed()
62.         .collect(Collectors.groupingBy(Function.identity(), Collectors.                          
63.         ()))
64.         .entrySet().forEach(System.out::println);
65. // 1=1
66. // 2=2
67. // 3=3
67. // 4=4
```

13. Советы и best practices

- 1. Если задачу не получается красиво решить стримами, не решайте её стримами.
- 2. Если задачу не получается красиво решить стримами, не решайте её стримами!
- 3. Если задача уже красиво решена не стримами, всё работает и всех всё устраивает, не перерешивайте её стримами!
- 4. В большинстве случаев нет смысла сохранять стрим в переменную. Используйте цепочку вызовов методов (method chaining).

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`

```
1. // Нечитабельно
2. Stream<Integer> stream = list.stream();
3. stream = stream.filter(x -> x > 2);
4. stream.forEach(System.out::println);
5. // Так лучше
6. list.stream()
7.         .filter(x -> x > 2)
8.         .forEach(System.out::println);
```

- 5. Старайтесь сперва отфильтровать стрим от ненужных элементов или ограничить его, а потом выполнять преобразования.

[\[копировать\]](#) `(javascript:void(0);)` [\[скачать\]](#) `(javascript:void(0);)`


```
1. // Лишние затраты
2. list.stream()
3.     .sorted()
4.     .filter(x -> x > 0)
5.     .forEach(System.out::println);
6. // Так лучше
7. list.stream()
8.     .filter(x -> x > 0)
9.     .sorted()
10.    .forEach(System.out::println);
```

6. Не используйте параллельные стримы везде, где только можно. Затраты на разбиение элементов, обработку в другом потоке и последующее их слияние порой больше, чем выполнение в одном потоке. Читайте об этом [здесь — When to use parallel streams](http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html) (<http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html>).

7. При использовании параллельных стримов, убедитесь, что нигде нет блокирующих операций или чего-то, что может помешать обработке элементов.

```
1. list.parallelStream()
2.     .filter(s -> isFileExists(hash(s)))
3.     ...
```



8. Если где-то в модели вы возвращаете копию списка или другой коллекции, то подумайте о замене на стримы. Например:

[копировать] (javascript:void(0);) [скачать] (javascript:void(0);)

```
1. // Было
2. class Model {
3.
4.     private final List<String> data;
5.
6.     public List<String> getData() {
7.         return new ArrayList<>(data);
8.     }
9. }
10.
11. // Стало
12. class Model {
13.
14.     private final List<String> data;
15.
16.     public Stream<String> dataStream() {
17.         return data.stream();
18.     }
19. }
```

Теперь есть возможность получить не только список `model.dataStream().collect(toList())` ; , но и множество, любую другую коллекцию, отфильтровать что-то, отсортировать и так далее. Оригинальный `List<String> data` так и останется нетронутым.

Если возникнут какие-либо вопросы, смело задавайте их в комментариях.

 +16 

 388086

 [aNNiMON\(/ablogs/?act=user&id=1\)](#)

Комментарии (21) (/ablogs/?act=comm&id=2778)