Странности Stream API

Тагир Валеев

Институт систем информатики СО РАН

Что это за чувак на сцене?

https://github.com/amaembo/streamex

StreamEx 0.6.0

Enhancing Java 8 Streams.

maven-central v0.6.0 | build passing | coverage 99%

This library defines four classes: `StreamEx`, `IntStreamEx`, `LongStreamEx`, `DoubleStreamEx` which are fully compatible with Java 8 stream classes and provide many additional useful methods. Also `EntryStream` class is provided which represents the stream of map entries and provides additional functionality for this case. Finally there are some new useful collectors defined in `MoreCollectors` class as well as primitive collectors concept.

Full API documentation is available here.

Take a look at the Cheatsheet for brief introduction to the StreamEx!

Before updating StreamEx check the migration notes and full list of changes.

StreamEx library main points are following:

- Shorter and convenient ways to do the common tasks.
- Better interoperability with older code.
- 100% compatibility with original JDK streams.
- Friendliness for parallel processing: any new feature takes the advantage on parallel streams as much as possible.
- Performance and minimal overhead. If StreamEx allows to solve the task using less code compared to standard Stream, it should not be significantly slower than the standard way (and sometimes it's even faster).

Что это за чувак на сцене?

tvaleev **Tagir F. Valeev**

Projects

jdk9 JDK 9 Project – Author

- JDK-8072727 Add variation of Stream.iterate() that's finite
- JDK-8136686 Collectors.counting can use Collectors.summingLong to reduce boxing
- JDK-8141630 Specification of Collections.synchronized* need to state traversal constraints
- JDK-8145007 Pattern splitAsStream is not late binding as required by the specification
- JDK-8146218 Add LocalDate.datesUntil method producing Stream<LocalDate>
- JDK-8147505 BaseStream.onClose() should not allow registering new handlers after stream is consumed
- JDK-8148115 Stream.findFirst for unordered source optimization
- JDK-8148250 Stream.limit() parallel tasks with ordered non-SUBSIZED source should short-circuit
- JDK-8148838 Stream.flatMap(...).spliterator() cannot properly split after tryAdvance()
- JDK-8148748 ArrayList.subList().spliterator() is not late-binding
- JDK-8151123 Collectors.summingDouble/averagingDouble unnecessarily call mapper twice

Что это за чувак на сцене?

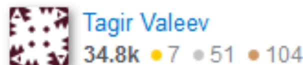
Help Center > Badges > Tags

java-stream

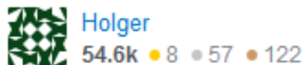
Earn at least 1000 total score for at least 200 non-community wiki answers in the java-stream tag. These users can single-handedly mark java-stream questions as duplicates and reopen them as needed.

Awarded 2 times

Awarded feb 4 at 5:50 to

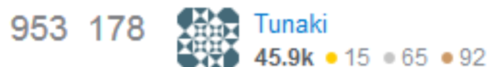


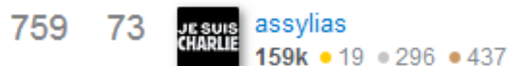
Awarded jan 26 at 3:06 to

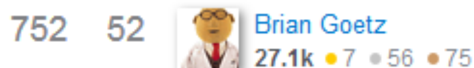


Top java-stream Answerers

All Time







```
LongStream.range(1, 100)
          .count();
```

```
LongStream.range(1, 100)
          .count();
>> 99
```

```java
LongStream.range(0, 1_000_000_000_000_000_000L)
          .count();
```

```java
LongStream.range(0, 1_000_000_000_000_000_000L)
                .count();
```

Java 9:

JDK-8067969 Optimize Stream.count for SIZED Streams

```
LongStream.range(0, 1_000_000_000_000_000_000L)
          .count();
>> 1000000000000000000
```

Характеристики

SIZED
SUBSIZED
SORTED
ORDERED

DISTINCT
NONNULL
IMMUTABLE
CONCURRENT

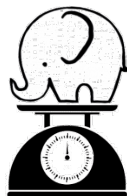
toArray()

```
IntStream.range(0, 100_000_000)
         .toArray();
```

```
IntStream.range(0, 100_000_000)
         .filter(x -> true)
         .toArray();
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.stream.SpinedBuffer$OfInt.newArray
    at java.util.stream.SpinedBuffer$OfInt.newArray
    at java.util.stream.SpinedBuffer$OfPrimitive.asPrimitiveArray
    at java.util.stream.Nodes$IntSpinedNodeBuilder.asPrimitiveArray
    at java.util.stream.Nodes$IntSpinedNodeBuilder.asPrimitiveArray
    at java.util.stream.IntPipeline.toArray
    at ru.javapoint.streamsamples.ToArray.main
```

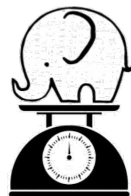
toArray()



```
IntStream.range(0, 100_000_000)        -Xmx560M     269ms
         .toArray();

IntStream.range(0, 100_000_000)        -Xmx1330M    795ms
         .filter(x -> true)
         .toArray();
```

.collect(toList())?

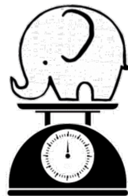
JDK-8072840 Add a method to Collector that returns a sized supplying mutable result container



```
IntStream.range(0, 10_000_000)      -Xmx320M     2.64±0.5s
   .boxed().collect(toList());

IntStream.range(0, 10_000_000)      -Xmx320M     2.63±0.5s
   .boxed().filter(x -> true)
   .collect(toList());
```

sorted()

full-barrier operation



```
IntStream.range(0, 100_000_000)
         .toArray();
```

-Xmx560M

```
IntStream.range(0, 100_000_000)
         .filter(x -> true)
         .toArray();
```

-Xmx1330M

```
IntStream.range(0, 100_000_000)
         .sorted().sum();
```

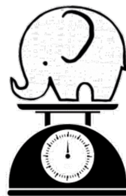
?

```
IntStream.range(0, 100_000_000)
         .filter(x -> true)
         .sorted().sum();
```

?

sorted()

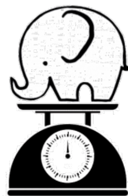
full-barrier operation



```
IntStream.range(0, 100_000_000)          -Xmx560M
         .toArray();

IntStream.range(0, 100_000_000)          -Xmx1330M
         .filter(x -> true)
         .toArray();

IntStream.range(0, 100_000_000)          -Xmx1M
         .sorted().sum();

IntStream.range(0, 100_000_000)          -Xmx1M
         .filter(x -> true)
         .sorted().sum();
```

sorted()

full-barrier operation



```
IntStream.range(0, 100_000_000)                -Xmx560M
         .toArray();

IntStream.range(0, 100_000_000)                -Xmx1330M
         .filter(x -> true)
         .toArray();

IntStream.range(0, 100_000_000)                -Xmx580M
         .map(x -> x).sorted().sum();

IntStream.range(0, 100_000_000)                -Xmx1330M
         .filter(x -> true)
         .map(x -> x).sorted().sum();
```

skip()

```
IntStream.range(0, 100_000_000)
         .toArray();
```

```
IntStream.range(0, 100_000_000)
         .skip(1)
         .toArray();
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.stream.SpinedBuffer$OfInt.newArray
    at java.util.stream.SpinedBuffer$OfInt.newArray
    at java.util.stream.SpinedBuffer$OfPrimitive.asPrimitiveArray
    at java.util.stream.Nodes$IntSpinedNodeBuilder.asPrimitiveArray
    at java.util.stream.Nodes$IntSpinedNodeBuilder.asPrimitiveArray
    at java.util.stream.IntPipeline.toArray
    at ru.javapoint.streamsamples.ToArraySkip.main
```

skip() и limit()

```
IntStream.range(0, 100_000_000)
         .toArray();
```

```
IntStream.range(0, 100_000_000)
         .skip(1)
         .toArray();
```

```
IntStream.range(0, 100_000_000)
         .limit(99_999_999)
         .toArray();
```

skip() и limit()

```
new Random().ints()
            .limit(100_000_000)
            .toArray();
```

```
new Random().ints(100_000_000)
            .toArray();
```

skip() и limit()

```
list.stream()
     .limit(2000)
     .skip(1000)
     .forEach(System.out::println);
```

```
list.subList(1000, 2000)
     .stream()
     .forEach(System.out::println);
```

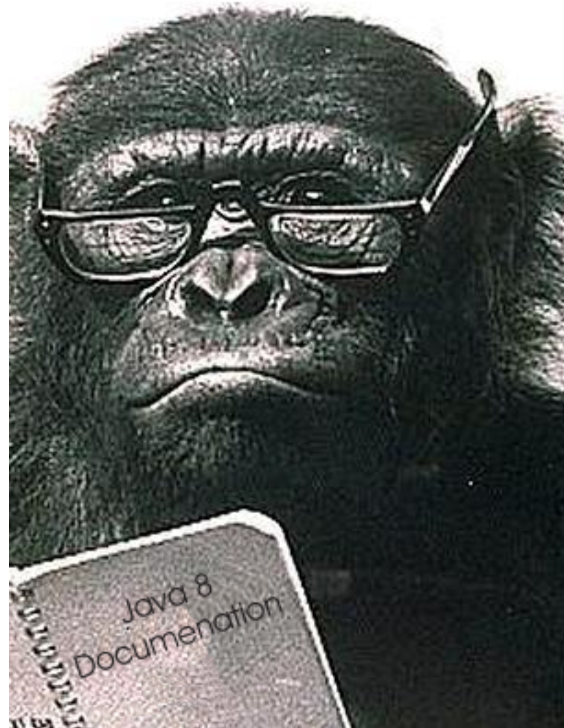
parallel().skip()

```
IntStream.range(0, 100_000_000)
         .skip(99_000_000)
         .sum();
```

104.5±4.4 ms

```
IntStream.range(0, 100_000_000)
         .parallel()
         .skip(99_000_000)
         .sum();
```

?


Java 8
Documenation

parallel().skip()

API Note:
While skip() is generally a cheap operation on sequential stream pipelines, it can be **quite expensive** on ordered **parallel** pipelines, especially for **large** values of n, since skip(n) is constrained to skip not just any *n* elements, but the *first n* elements in the encounter order.

parallel().skip()

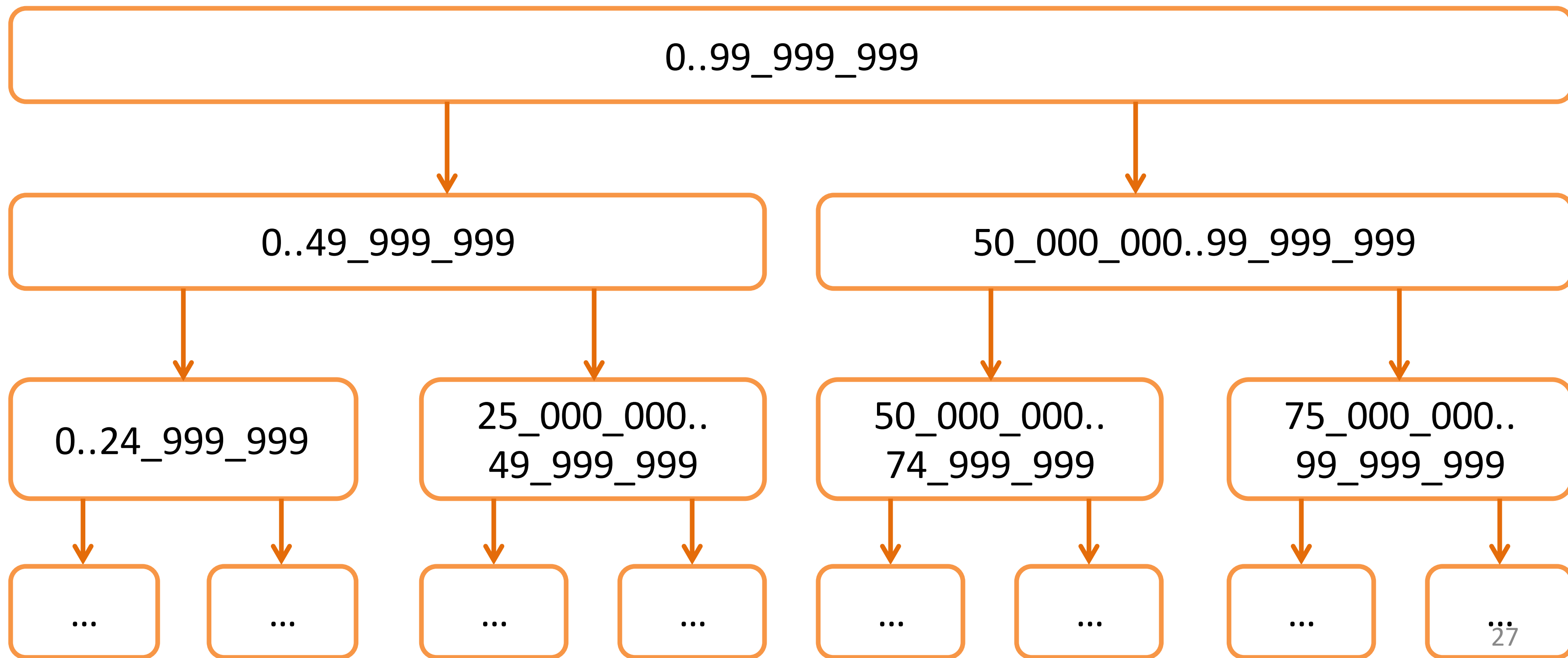
```
IntStream.range(0, 100_000_000)
         .skip(99_000_000)
         .sum();
```

104.5±4.4 ms

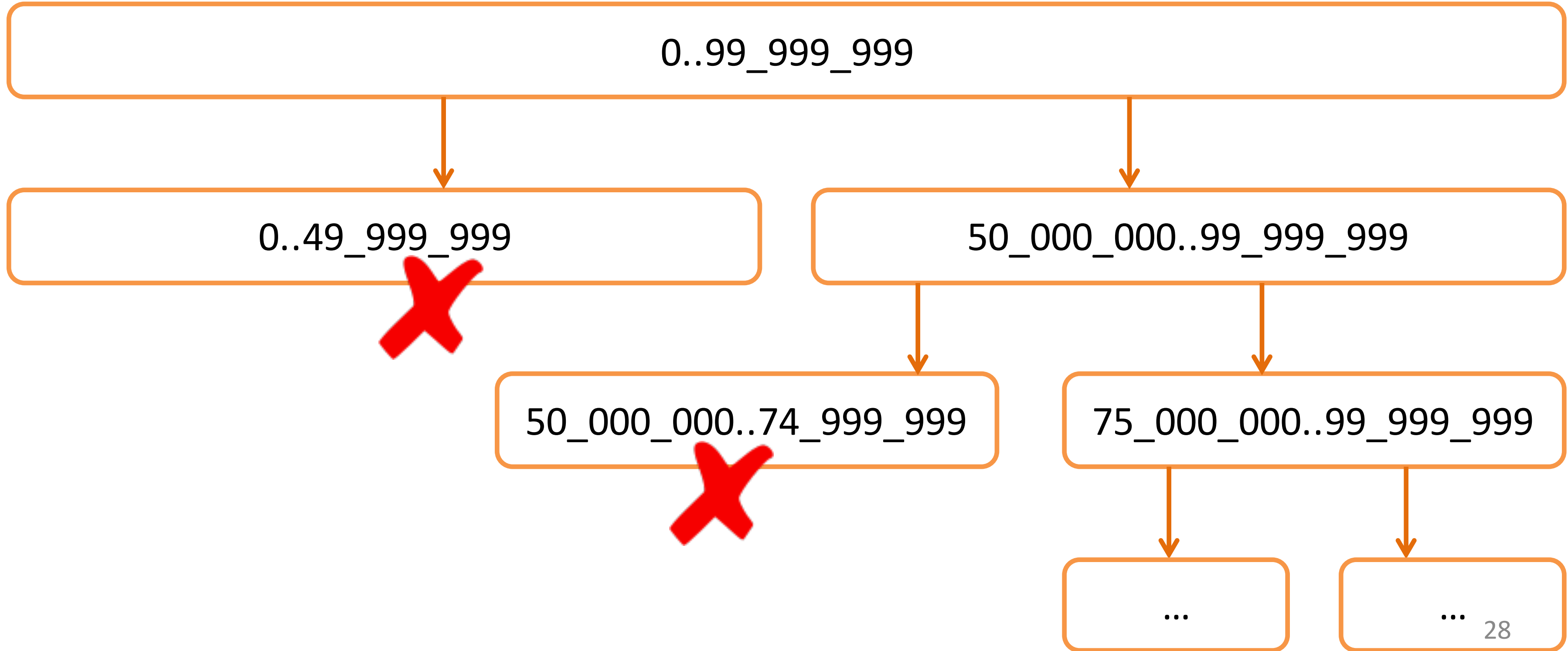
```
IntStream.range(0, 100_000_000)
         .parallel()
         .skip(99_000_000)
         .sum();
```

1.4±0.2 ms
(74.6×)

parallel(): trySplit() (SIZED+SUBSIZED)



parallel().skip() (SIZED+SUBSIZED)



Характеристики

SIZED
SUBSIZED
SORTED
ORDERED

DISTINCT
NONNULL
IMMUTABLE
CONCURRENT

distinct() и ordering

```
List<Integer> input = new Random(1)
        .ints(10_000_000, 0, 10)
        .boxed().collect(Collectors.toList());

input.stream()
     .distinct()
     .collect(Collectors.toList());
```

85.4 ± 0.7 ms

distinct() и ordering

```
input.stream()
      .parallel()
      .distinct()
      .collect(Collectors.toList());
```

30.5 ± 1.7 ms **(2.8×)**

parallel().distinct()

API Note:
Preserving stability for distinct() in parallel pipelines is relatively expensive (requires that the operation act as a full barrier, with substantial buffering overhead), and stability is often not needed. Using an **unordered** stream source (such as generate(Supplier)) or removing the ordering constraint with BaseStream.**unordered()** may result in **significantly more efficient** execution for distinct() in parallel pipelines, if the semantics of your situation permit.

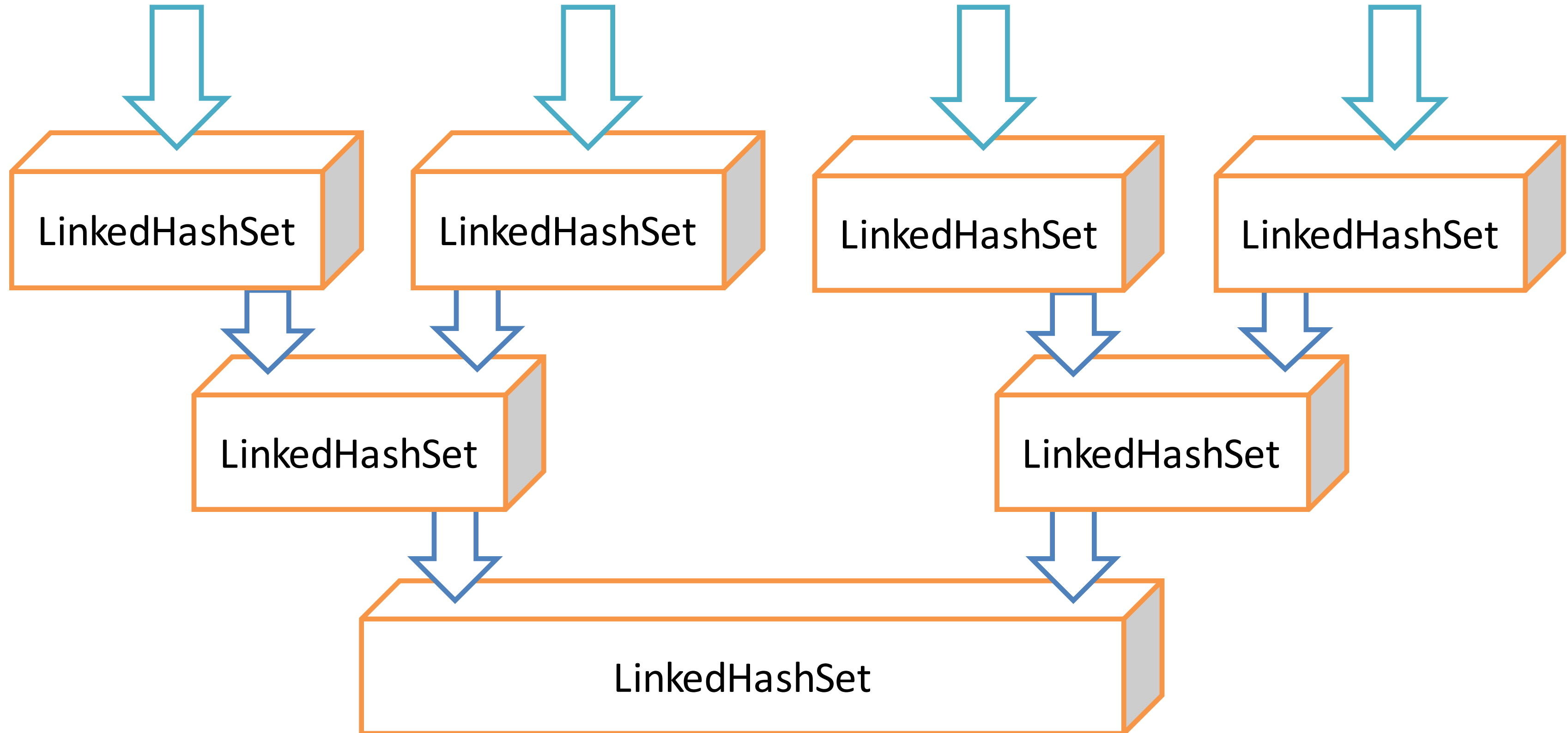
distinct() и ordering

```
input.stream()
      .parallel()
      .unordered()
      .distinct()
      .collect(Collectors.toList());
```

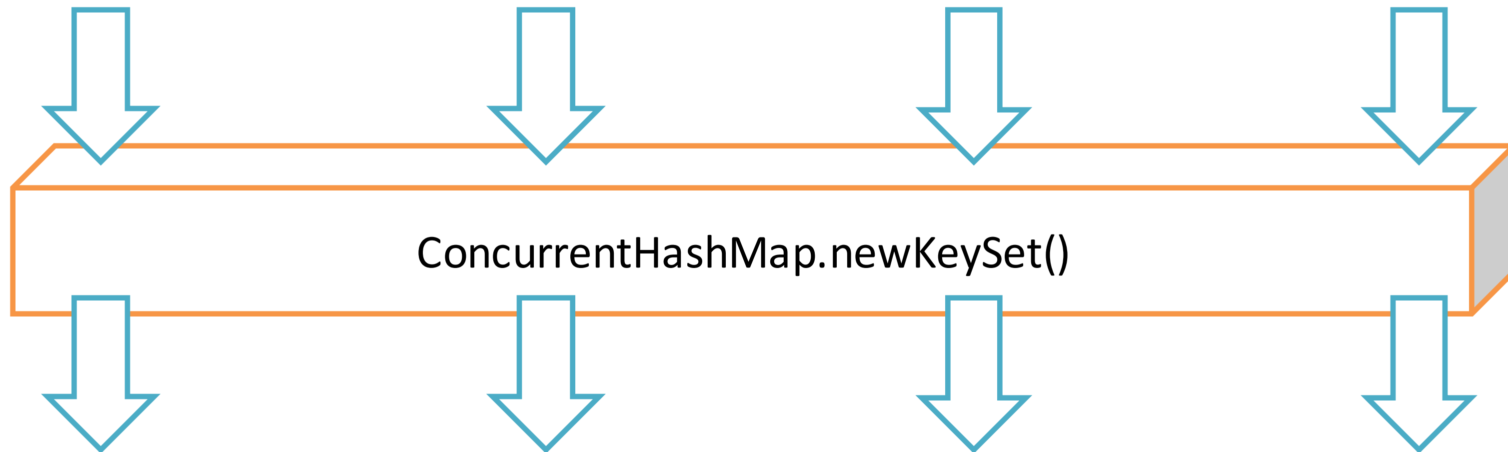
distinct() и ordering

```
Sequential              85.4 ± 0.7 ms
Parallel                30.5 ± 1.7 ms (2.8×)
Parallel unordered     249.0 ± 1.5 ms (0.34×)
```

distinct() ordered parallel



distinct() unordered parallel



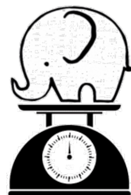
Характеристики

SIZED
SUBSIZED
SORTED
ORDERED

DISTINCT
NONNULL
IMMUTABLE
CONCURRENT

Stream.concat()

```
IntStream s1 = IntStream.range(0, 50_000_000);
IntStream s2 = IntStream.range(50_000_000, 100_000_000);
```



```
IntStream.concat(s1, s2).toArray();      -Xmx580M
```

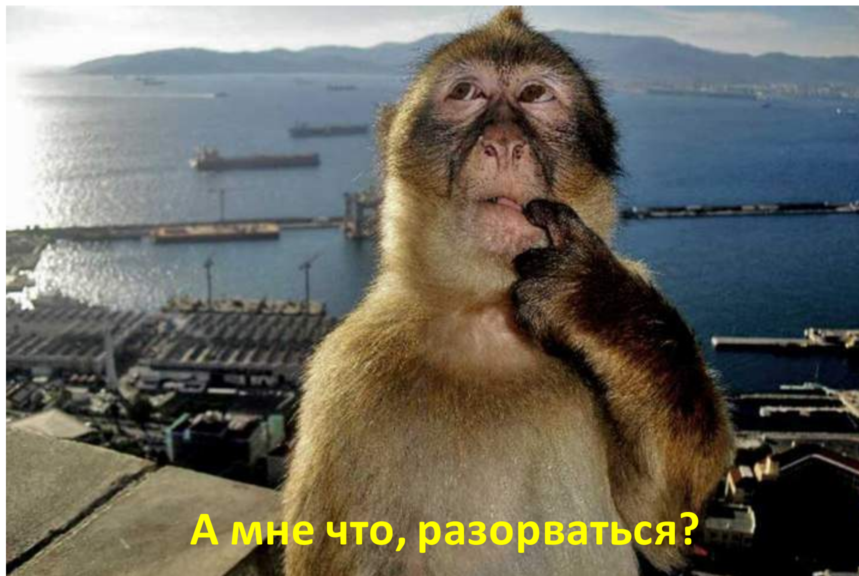
Промежуточные операции (красивые)

map(), filter(), flatMap(), …

Терминальные операции (умные)

forEach(), reduce(), count(), …

Stream.concat()



concat() и parallel()

```
Integer[] aIntegers = { 101, 102, ..., 200 };
Integer[] bIntegers = { 201, 202, ..., 300 };

Stream<Integer> a = Arrays.stream(aIntegers);
Stream<Integer> b = Arrays.stream(bIntegers);
List<String> list = Stream.concat(a, b)
    .map(num -> String.valueOf(num).replaceAll("((((.*)*)*)*)*!", ""))
    .collect(Collectors.toList());
```

868.9 ± 10.6 ms

concat() и parallel()

```
Integer[] aIntegers = { 101, 102, ..., 200 };
Integer[] bIntegers = { 201, 202, ..., 300 };

Stream<Integer> a = Arrays.stream(aIntegers);
Stream<Integer> b = Arrays.stream(bIntegers);
List<String> list = Stream.concat(a, b)
    .parallel()
    .map(num -> String.valueOf(num).replaceAll("((((.*)*)*)*)*!", ""))
    .collect(Collectors.toList());
```

227.9 ± 3.5 ms (**3.8×**)

concat() и parallel()

```
Stream<Integer> a = Arrays.stream(aIntegers);
Stream<Integer> b = Arrays.stream(bIntegers);
List<String> list = Stream.concat(a, b)
    .parallel()
    .map(num -> String.valueOf(num).replaceAll(...))
    .collect(Collectors.toList());
```

227.9
± 3.5 ms
(3.8×)

```
Stream<Integer> a = Arrays.stream(aInts).boxed();
Stream<Integer> b = Arrays.stream(bInts).boxed();
List<String> list = Stream.concat(a, b)
    .parallel()
    .map(num -> String.valueOf(num).replaceAll(...))
    .collect(Collectors.toList());
```

?

concat() и parallel()

```
Stream<Integer> a = Arrays.stream(aIntegers);
Stream<Integer> b = Arrays.stream(bIntegers);
List<String> list = Stream.concat(a, b)
    .parallel()
    .map(num -> String.valueOf(num).replaceAll(...))
    .collect(Collectors.toList());
```

227.9
± 3.5 ms
(3.8×)

```
Stream<Integer> a = Arrays.stream(aInts).boxed();
Stream<Integer> b = Arrays.stream(bInts).boxed();
List<String> list = Stream.concat(a, b)
    .parallel()
    .map(num -> String.valueOf(num).replaceAll(...))
    .collect(Collectors.toList());
```

437.8
± 5.2 ms
(1.98×)

concat()

1. Пусть A = aStream.spliterator(), B = bStream.spliterator()
2. Создать новый spliterator из этих двух:
   - **tryAdvance**: вызывать A.tryAdvance(),
     а если там кончилось, то B.tryAdvance().
   - **forEachRemaining**: вызвать A.forEachRemaining(),
     затем B.forEachRemaining().
   - **trySplit**: разделить назад на A и B.
3. Создать новый stream по новому сплитератору.
4. Повесить на onClose вызов aStream.close() и bStream.close()

Stream.spliterator()

1. Нет промежуточных операций? Вернём исходный сплитератор.
2. Есть промежуточные операции? Создадим новый WrappingSpliterator:
   - **forEachRemaining()**: ≈ Stream.forEach()
   - **tryAdvance()**: вызвать tryAdvance у источника и собрать в буфер, что накопилось, а потом идти по буферу
   - **trySplit()**: если исходный стрим параллельный, вызвать trySplit() у источника и обернуть результат в такой же WrappingSpliterator

concat() и parallel()

```
Stream<Integer> a = Arrays.stream(aIntegers);
Stream<Integer> b = Arrays.stream(bIntegers);
List<String> list = Stream.concat(a, b)
    .parallel()
    .map(num -> String.valueOf(num).replaceAll(...))
    .collect(Collectors.toList());
```

227.9 ± 3.5 ms **(3.8×)**

```
Stream<Integer> a = Arrays.stream(aInts).boxed();
Stream<Integer> b = Arrays.stream(bInts).boxed();
List<String> list = Stream.concat(a, b)
    .parallel()
    .map(num -> String.valueOf(num).replaceAll(...))
    .collect(Collectors.toList());
```

437.8 ± 5.2 ms **(1.98×)**

concat() и parallel()

```
Stream<Integer> a = Arrays.stream(aInts).boxed().parallel();
Stream<Integer> b = Arrays.stream(bInts).boxed().parallel();
List<String> list = Stream.concat(a, b)
   .map(num -> String.valueOf(num).replaceAll(...))
   .collect(Collectors.toList());
```

222.1 ± 2.3 ms (**3.9×**)

concat() и ordering

```
List<Integer> a = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
List<Integer> b = Arrays.asList();
Stream.concat(a.parallelStream(), b.parallelStream())
     .filter(x -> x % 2 == 0)
     .limit(3)
     .forEachOrdered(System.out::println);
>> 2
>> 4
>> 6
```

concat() и ordering

```
List<Integer> a = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
List<Integer> b = Collections.emptyList();
Stream.concat(a.parallelStream(), b.parallelStream())
    .filter(x -> x % 2 == 0)
    .limit(3)
    .forEachOrdered(System.out::println);
>> 2
>> 6
>> 10
```

flatMap()

```
Stream<Integer> a = ...;
Stream<Integer> b = ...;
Stream<Integer> c = ...;
Stream<Integer> d = ...;
Stream<Integer> res = Stream.concat(
    Stream.concat(Stream.concat(a, b), c), d);

Stream<Integer> res = Stream.of(a, b, c, d)
    .flatMap(Function.identity());
```

concat() или flatMap()?

```
Stream<Integer> a = Arrays.stream(aInts).boxed().parallel();
Stream<Integer> b = Arrays.stream(bInts).boxed().parallel();
List<String> list = Stream.of(a, b)
    .flatMap(Function.identity())
    .parallel()
    .map(num -> String.valueOf(num).replaceAll(...))
    .collect(Collectors.toList()); // 444.8 ± 7.3 ms (1.95x)
```

concat() или flatMap()?

```
IntStream s1 = IntStream.range(0, 50_000_000);
IntStream s2 = IntStream.range(50_000_000, 100_000_000);

IntStream.concat(s1, s2).toArray(); // -Xmx580M

Stream.of(s1, s2).flatMapToInt(Function.identity())
      .toArray(); // -Xmx1330M
```

flatMap() и short-circuiting

```
IntStream s1 = IntStream.range(0, 50_000_000);
IntStream s2 = IntStream.range(50_000_000, 100_000_000);

IntStream.concat(s1, s2)
  .filter(x -> x > 2).findFirst(); //      0.13 µs

Stream.of(s1, s2).flatMapToInt(Function.identity())
  .filter(x -> x > 2).findFirst(); // 301051 µs
```

flatMap() и tryAdvance()

```
Stream<Integer> s =
    IntStream.range(0, 1_000_000_000).boxed();
s.spliterator().tryAdvance(System.out::println);
>> 0
```

flatMap() и tryAdvance()

```
Stream<Integer> s = IntStream.of(1_000_000_000)
    .flatMap(x -> IntStream.range(0, x)).boxed();
s.spliterator().tryAdvance(System.out::println);
```

```
java.lang.OutOfMemoryError: Java heap space
  at java.util.stream.SpinedBuffer.ensureCapacity
  at java.util.stream.SpinedBuffer.increaseCapacity
  at java.util.stream.SpinedBuffer.accept
  at java.util.stream.IntPipeline$4$1.accept
  at java.util.stream.IntPipeline$7$1.lambda$accept$198
  at java.util.stream.IntPipeline$7$1$$Lambda$7/1831932724.accept
  at java.util.stream.Streams$RangeIntSpliterator.forEachRemaining
  at java.util.stream.IntPipeline$Head.forEach
  at java.util.stream.IntPipeline$7$1.accept
  at java.util.stream.Streams$IntStreamBuilderImpl.tryAdvance
  at java.util.Spliterator$OfInt.tryAdvance
  ...
  at java.util.stream.StreamSpliterators$WrappingSpliterator.tryAdvance
  at ru.javapoint.streamsamples.FlatMapTryAdvance.main
```

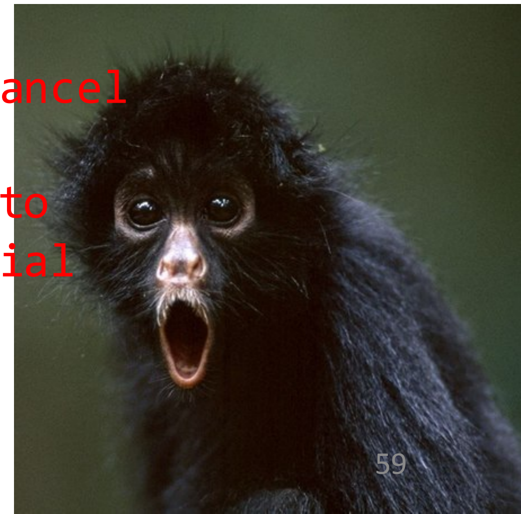
flatMap() и tryAdvance()

```
Stream<Integer> s = IntStream.of(1_000_000_000)
    .flatMap(x -> IntStream.range(0, x)).boxed();
s.spliterator().tryAdvance(System.out::println);
```

flatMap() и concat()

```
IntStream s = IntStream.of(1_000_000_000)
  .flatMap(x -> IntStream.range(0, x));

s.sum(); ✓

s.findFirst(); ✓    // non short-circuit

IntStream.concat(s, IntStream.of(1)).sum(); ✓

IntStream.concat(s, IntStream.of(1)).findFirst();
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at java.util.stream.SpinedBuffer$OfInt.newArray
  at java.util.stream.SpinedBuffer$OfInt.newArray
  at java.util.stream.SpinedBuffer$OfPrimitive.ensureCapacity
  at java.util.stream.SpinedBuffer$OfPrimitive.increaseCapacity
  at java.util.stream.SpinedBuffer$OfPrimitive.preAccept
  at java.util.stream.SpinedBuffer$OfInt.accept
  ...
  at java.util.stream.Streams$ConcatSpliterator$OfInt.tryAdvance
  at java.util.stream.IntPipeline.forEachWithCancel
  at java.util.stream.AbstractPipeline.copyIntoWithCancel
  at java.util.stream.AbstractPipeline.copyInto
  at java.util.stream.AbstractPipeline.wrapAndCopyInto
  at java.util.stream.FindOps$FindOp.evaluateSequential
  at java.util.stream.AbstractPipeline.evaluate
  at java.util.stream.IntPipeline.findFirst
  at ru.javapoint.streamsamples.ConcatFlat.main
```



flatMap() vs concat()

	concat()	flatMap()
Сохраняет SIZED	✓	✗
Short-circuiting	✓	✗
Memory-friendly tryAdvance()	✓	✗
Полноценный параллелизм	±	✗
Всегда сохраняет порядок	✗	✓
Многократная конкатенация	✗	✓

Всё же concat()?

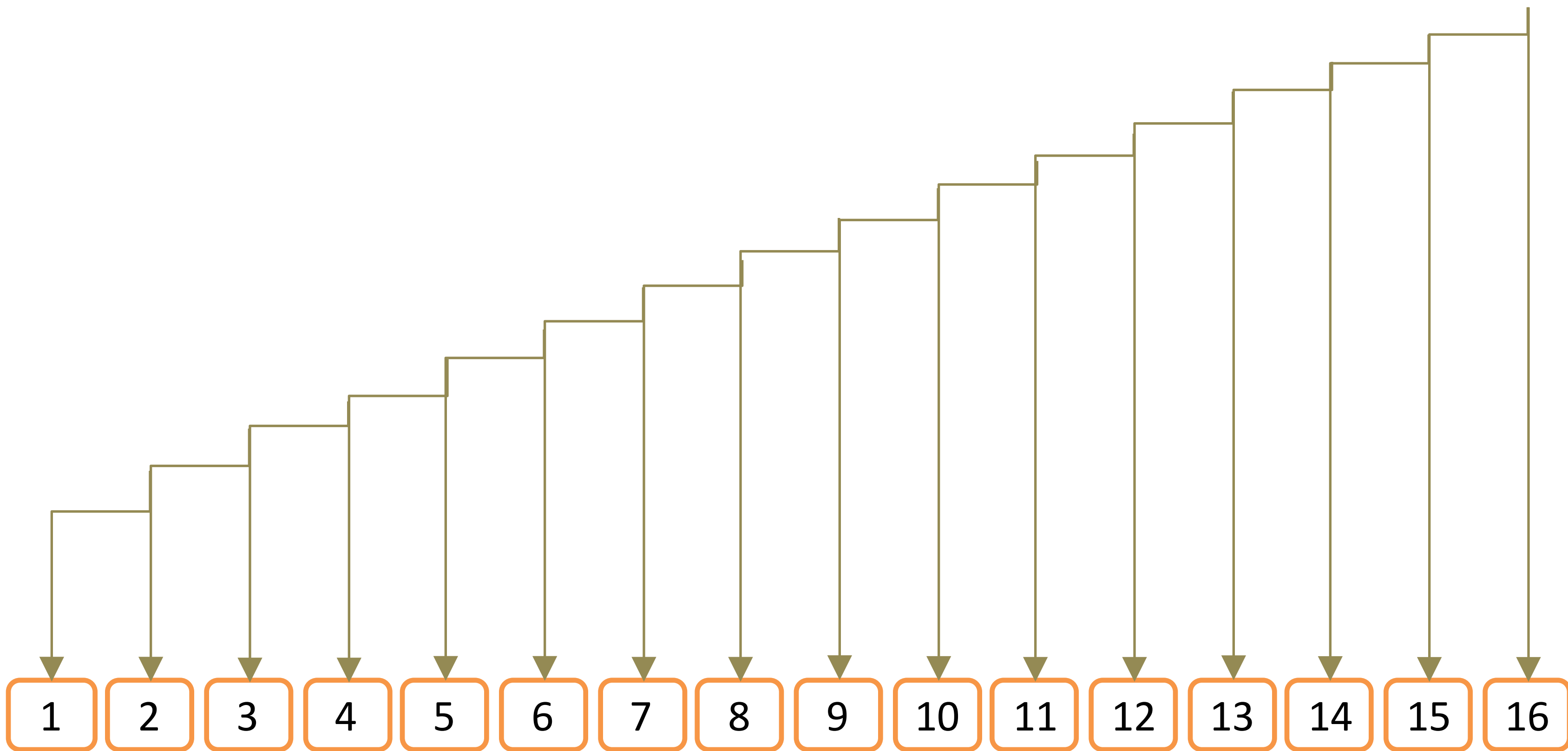
```
Object[][] data = new Object[20000][4000];

Object[] flat = Arrays.stream(data)
        .flatMap(Arrays::stream).toArray();
>> java.lang.OutOfMemoryError
```

Всё же concat()?

```
Object[][] data = new Object[20000][4000];

Object[] flat = Arrays.stream(data)
        .flatMap(Arrays::stream).toArray();
>> java.lang.OutOfMemoryError

Stream<Object> s = Arrays.stream(data)
    .map(Arrays::stream).reduce(Stream::concat)
    .orElse(Stream.empty());
Object[] flat = s.toArray();
```

```
Exception in thread "main" java.lang.StackOverflowError
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.forEachRemaining
  at java.util.stream.Streams$ConcatSpliterator.f
  at java.util.stream.Streams$ConcatSpliterator.f
  at java.util.stream.Streams$ConcatSpliterator.f
  at java.util.stream.Streams$ConcatSpliterator.f
  at java.util.stream.Streams$ConcatSpliterator.f
  at java.util.stream.Streams$ConcatSpliterator.f
  ...
```

reduce(Stream::concat)



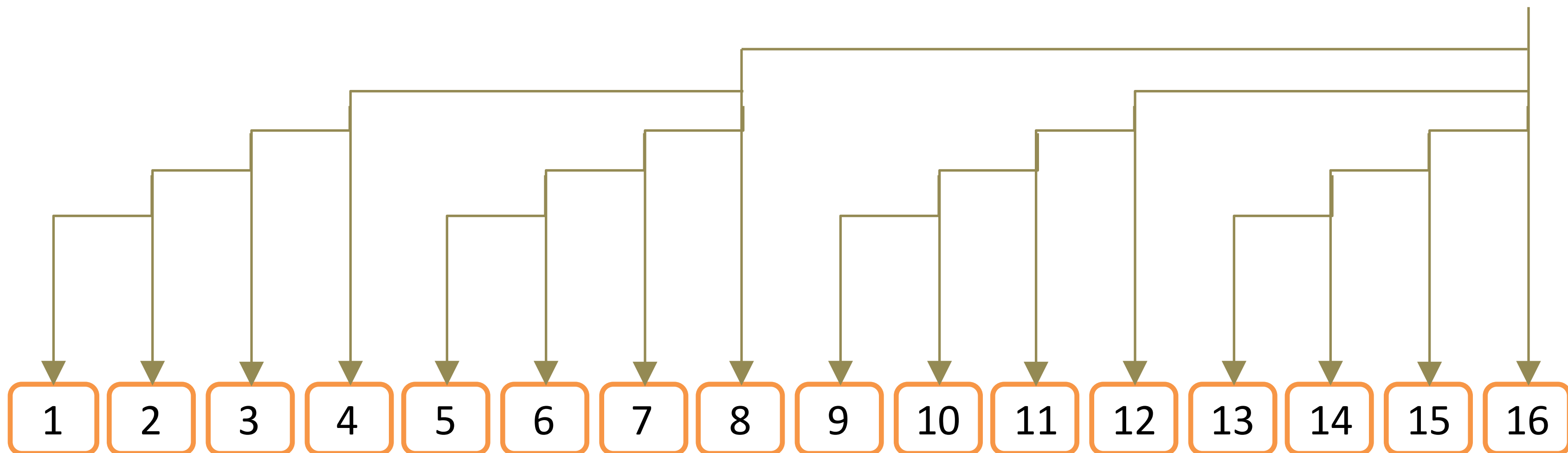
Распараллелим!

```
Object[][] data = new Object[20000][4000];

Stream<Object> s = Arrays.stream(data)
    .parallel()
    .map(Arrays::stream)
    .reduce(Stream::concat)
    .orElse(Stream.empty());
Object[] flat = s.toArray();
```

parallel().reduce(Stream::concat)

```
.reduce(Predicate::or)
.reduce(Predicate::and)
.reduce(Function::andThen)
```

https://habrahabr.ru/post/255813/



Stream и Iterator

iterator()

for vs forEach()

	for	forEach()
Появился	Java 1.5	Java 1.8
Нормальная отладка	✓	✗
Короткие стектрейсы	✓	✗
Checked exceptions	✓	✗
Изменяемые переменные	✓	✗
Досрочный выход по break	✓	✗
Параллелизм (всё равно не нужен)	✗	✓

iterator()

Joshua Bloch
@joshbloch

Читаю

It seems HORRIBLY BROKEN that BaseStream doesn't extend Iterable, given that it has an iterator() method. What am I missing?

РЕТВИТОВ	ОТМЕТКА «НРАВИТСЯ»
11	31

18:21 - 2 марта 2016 г.

iterator()

```
Stream<String> s = Stream.of("a", "b", "c");
for(String str : (Iterable<String>)s::iterator) {
    System.out.println(str);
}
```

Joshua Bloch
@joshbloch

Читаю

@stuartmarks Wow, that's repulsive! They're making me say "mother may I" to use a for-each loop, in a totally non-intuitive way. Yecch!

Показать перевод

ОТМЕТКИ «НРАВИТСЯ»
4

iterator()

```
Stream<String> s = Stream.of("a", "b", "c");
for(String str : (Iterable<String>)s::iterator) {
    System.out.println(str);
}

// Joshua Bloch approves
for(String str : StreamEx.of("a", "b", "c")) {
    System.out.println(str);
}
```

iterator()

```
IntStream s = IntStream.of(1_000_000_000)
  .flatMap(x -> IntStream.range(0, x));
for(int i : (Iterable<Integer>)s::iterator) {
    System.out.println(i);
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at j.u.s.SpinedBuffer$OfInt.newArray
  at j.u.s.SpinedBuffer$OfInt.newArray
  at j.u.s.SpinedBuffer$OfPrimitive.ensureCapacity
  at j.u.s.SpinedBuffer$OfPrimitive.increaseCapacity
  at j.u.s.SpinedBuffer$OfPrimitive.preAccept
  at j.u.s.SpinedBuffer$OfInt.accept
  at j.u.s.IntPipeline$7$1.lambda$accept$198
  ...
  at j.u.s.StreamSpliterators$AbstractWrappingSpliterator.fillBuffer
  at j.u.s.StreamSpliterators$AbstractWrappingSpliterator.doAdvance
  at j.u.s.StreamSpliterators$IntWrappingSpliterator.tryAdvance
  at j.u.Spliterators$2Adapter.hasNext
  at ru.javapoint.streamsamples.IterableWorkaround.main
```

Stream из итератора

```
StreamSupport.stream(
    Spliterators.spliteratorUnknownSize(
        iterator, Spliterator.ORDERED), false);
```

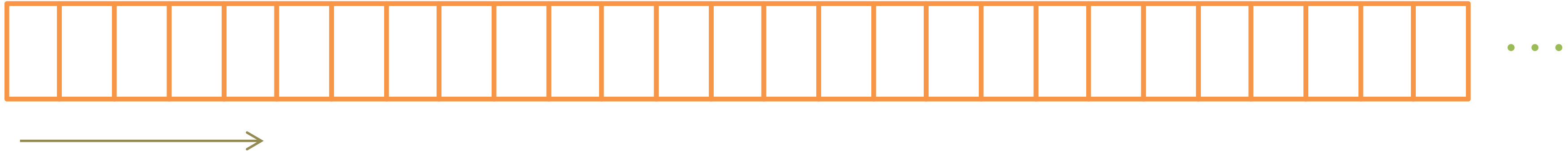
Stream из итератора

```
StreamSupport.stream(
    Spliterators.spliteratorUnknownSize(
        iterator, Spliterator.ORDERED), false);

StreamSupport.stream(
   ((Iterable<String>)() -> iterator).spliterator(),
   false);
```

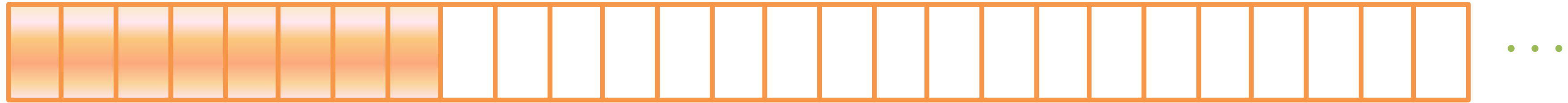
Stream из итератора

```
Files.find();
Files.lines();
Files.list();
Files.walk();
BufferedReader.lines();
Pattern.splitAsStream();
```

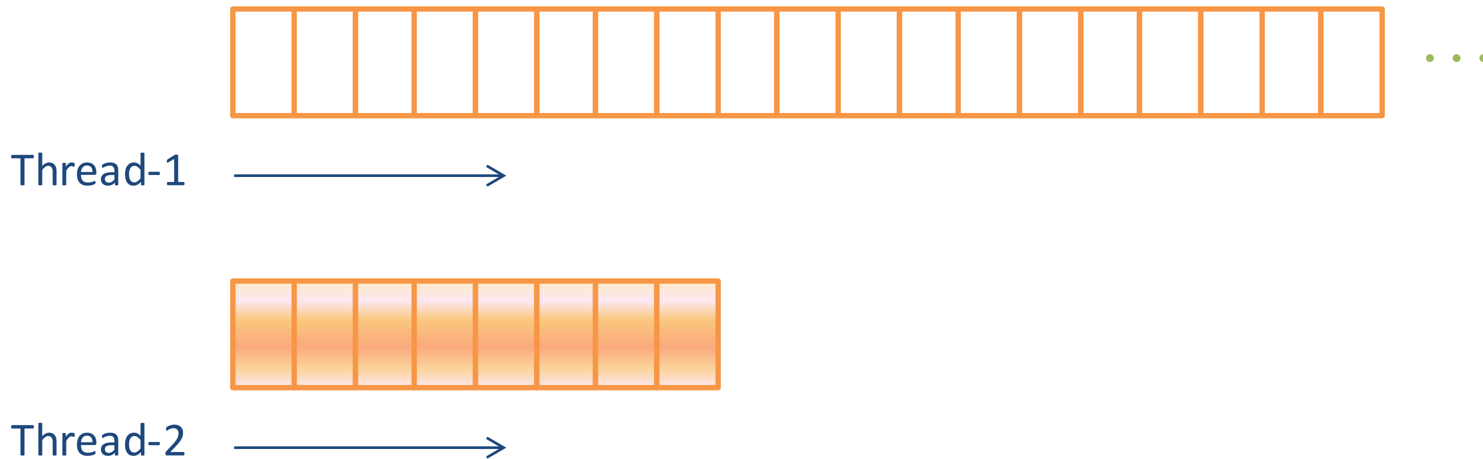
Как распараллелить последовательное?



Как распараллелить последовательное?



Как распараллелить последовательное?



Files.list()

```
List<Record> list = Files.list(root)
     .map(path -> parse(path))
     .collect(Collectors.toList());
```

204 ms

Files.list().parallel()

```
List<Record> list = Files.list(root)
     .parallel()
     .map(path -> parse(path))
     .collect(Collectors.toList());
```

202 ms

Files.list().parallel()

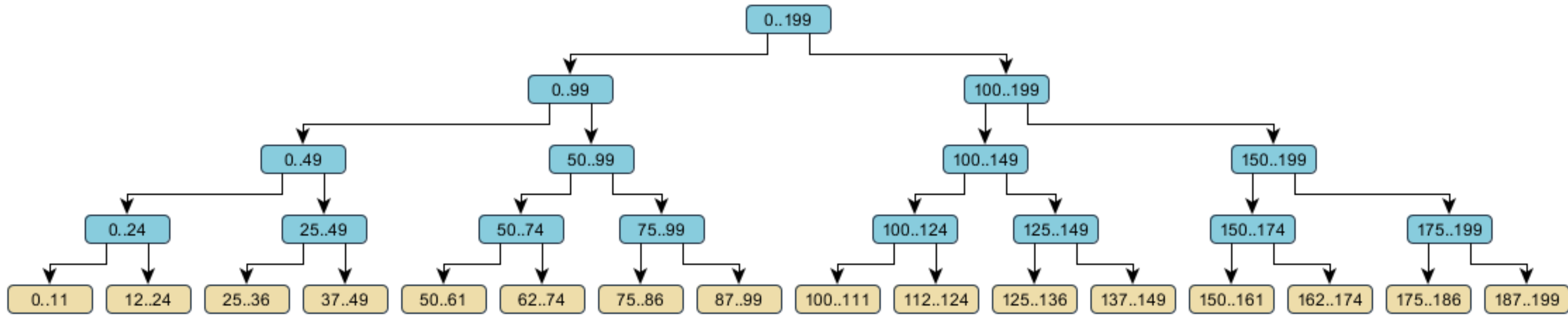
```
List<Record> list = Files.list(root)
     .parallel()
     .sorted(comparingLong(p -> p.toFile().length()))
     .map(path -> parse(path))
     .collect(Collectors.toList());     57.5 ms (3.5×)
```

StreamTools

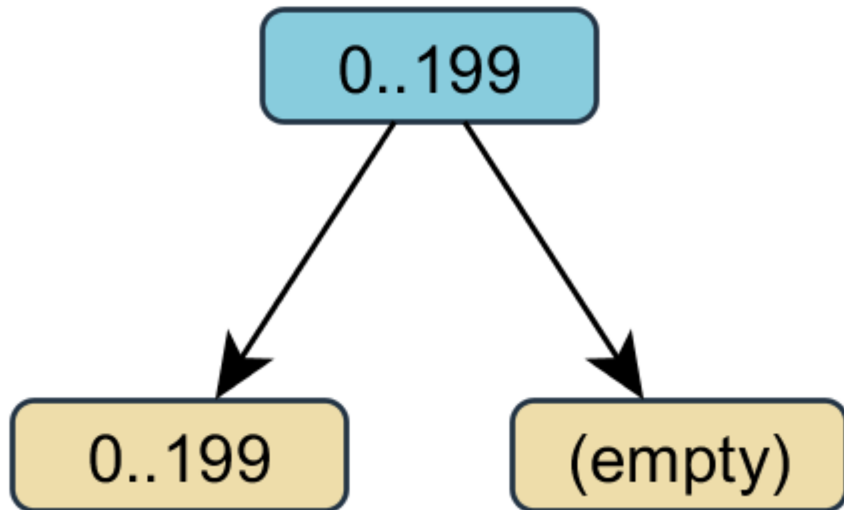
https://github.com/amaembo/streamtools

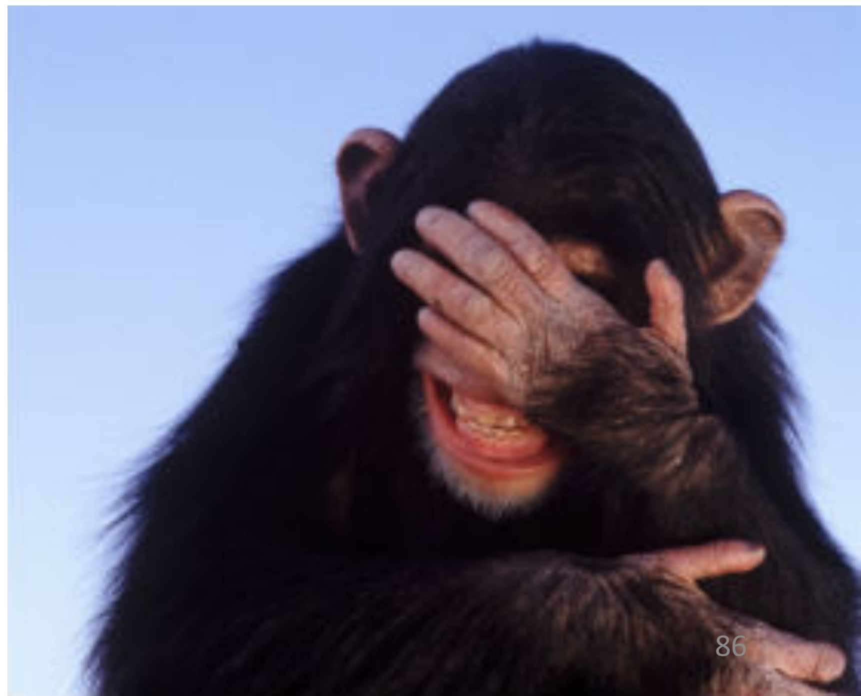
```
SplitTree tree = SplitTree.inspect(IntStream.range(0, 200));
try(OutputStream os = Files.newOutputStream(
        Paths.get("range.xgml"))) {
    new XGMLFormatter().nodeWidth(60).nodeHeight(20)
        .nodeFormat("%f..%l", "(empty)")
        .writeTo(tree, os);
}
```

IntStream.range(0, 200)

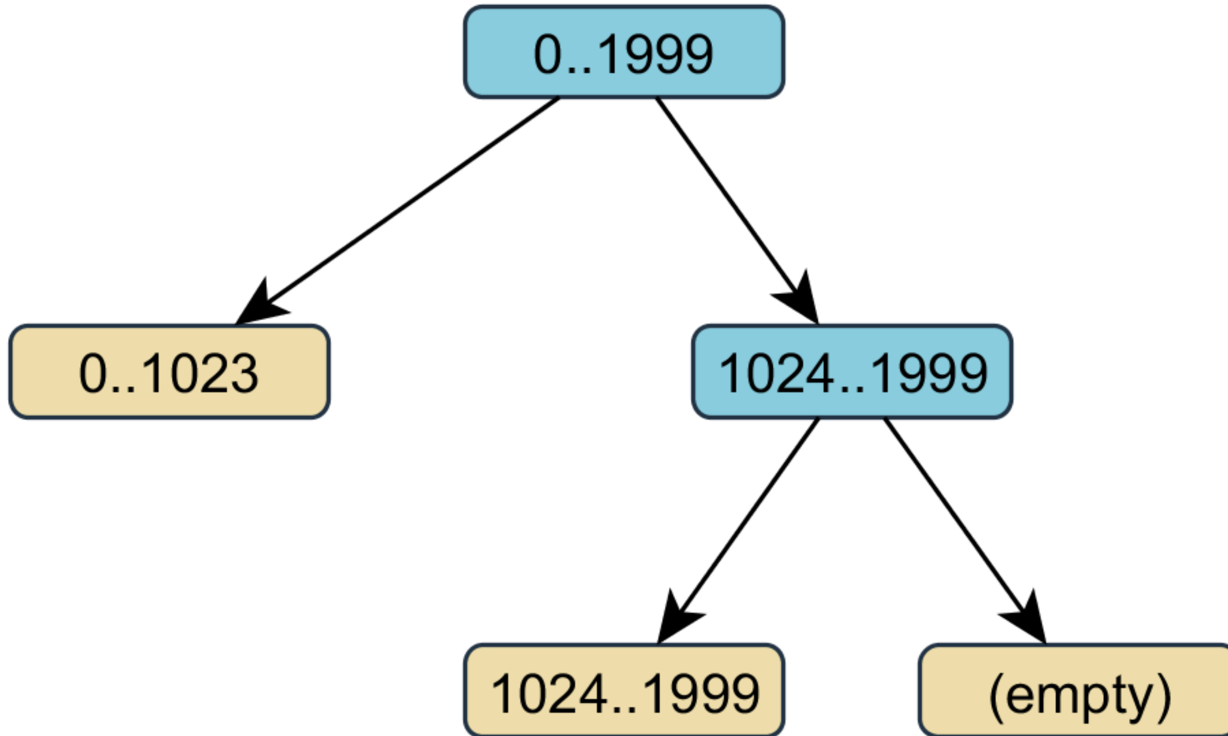


Stream из итератора (200)

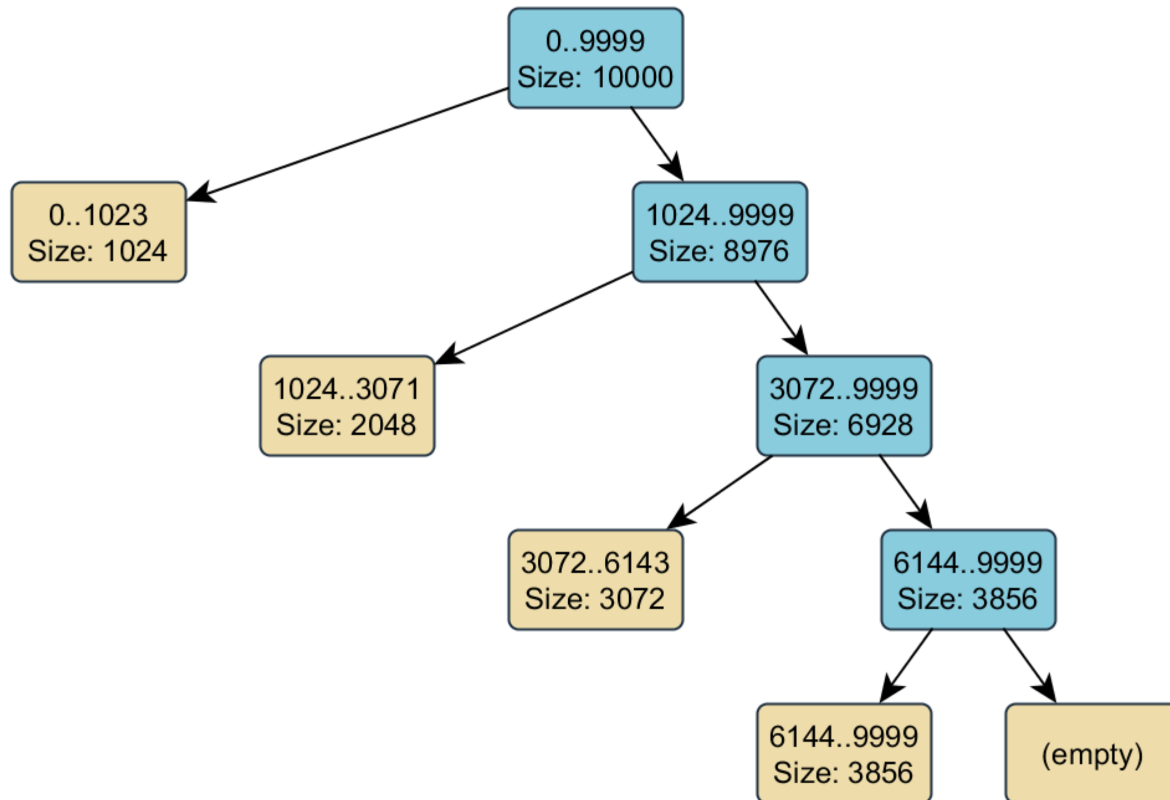


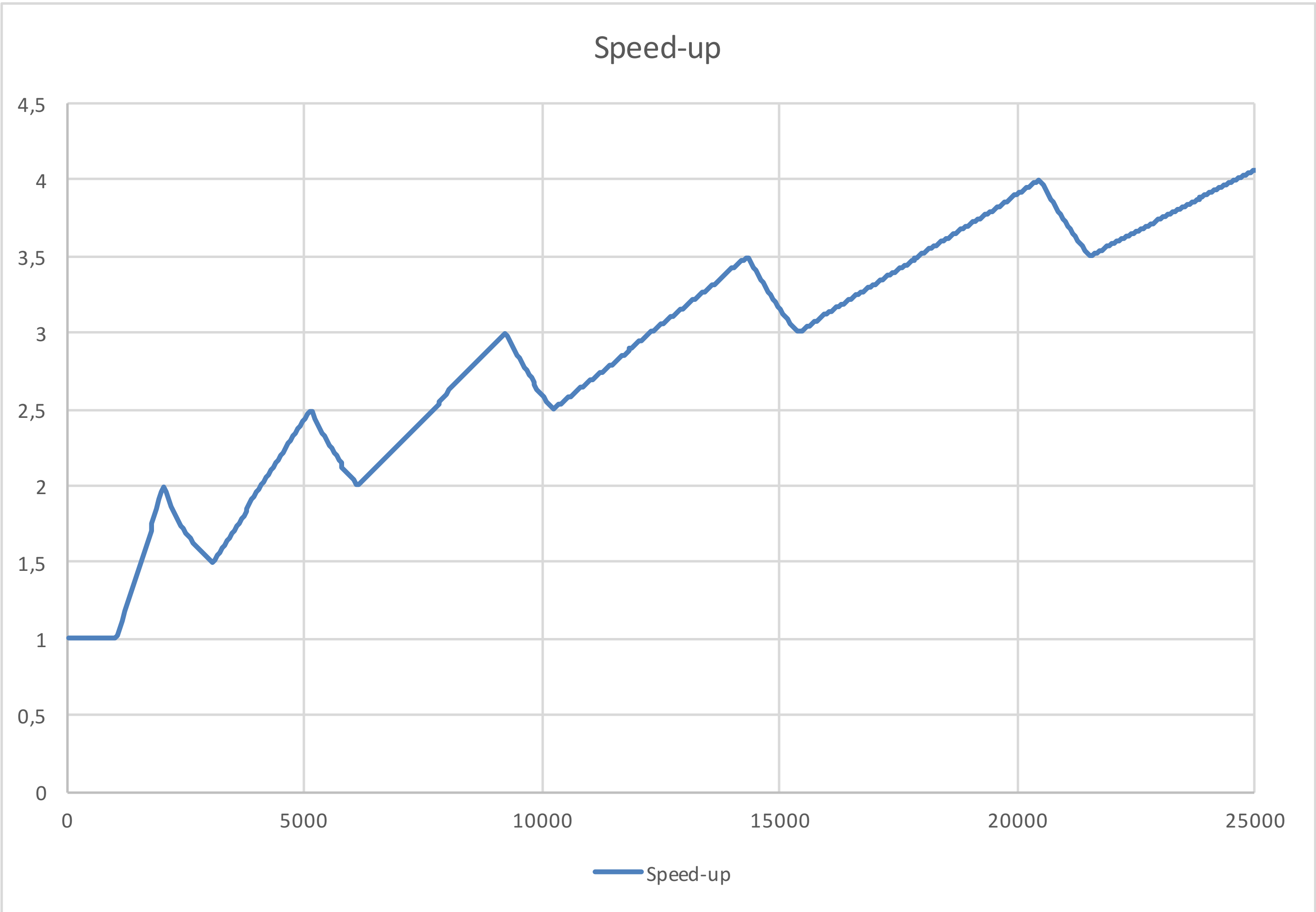


Stream из итератора (2000)



Stream из итератора (10000)

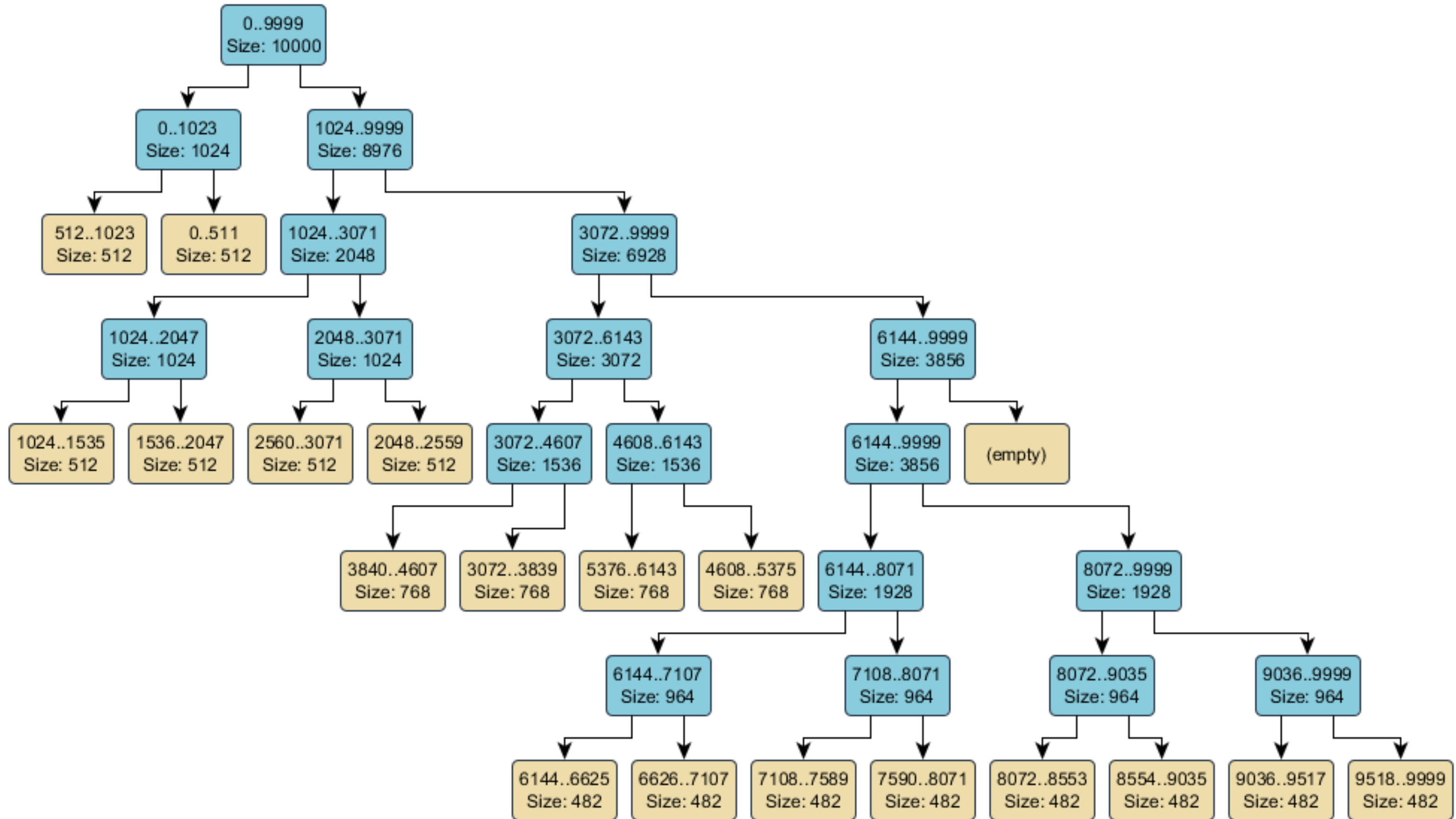



Speed-up
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
0
5000
10000
15000
20000
25000
Speed-up

Stream из итератора

```
StreamSupport.stream(
    Spliterators.spliterator(
        iterator, size, Spliterator.ORDERED), false);
```

SIZED-Stream из итератора (10000)



Files.list()

```
List<Record> list = Files.list(root)
    .collect(Collectors.toList())
    .parallelStream()
    .map(path -> parse(path))
    .collect(Collectors.toList());
```

Спасибо за внимание

https://twitter.com/tagir_valeev

https://github.com/amaembo

https://habrahabr.ru/users/lany