Причуды Stream API

Тагир Валеев

JetBrains


Against The Stream
Buddhist Meditation Society
ATBMS

JBreak 2016

Stream API: рекомендации лучших собаководов

http://www.slideshare.net/tvaleev/stream-api-59769165

https://www.youtube.com/watch?v=vxikpWnnnCU

JPoint 2016

Странности Stream API

http://www.slideshare.net/tvaleev/stream-api-61280104

https://www.youtube.com/watch?v=TPHMyVyktsw

Tagir Valeev
@tagir_valeev

Делаю новый доклад про Stream API на #jokerconf. Что вы хотите видеть?

62% Больше жизненных примеров

27% Больше параллельного ада

11% Плевать

112 голосов • Окончательные итоги

Причуда №1

Stream<T>

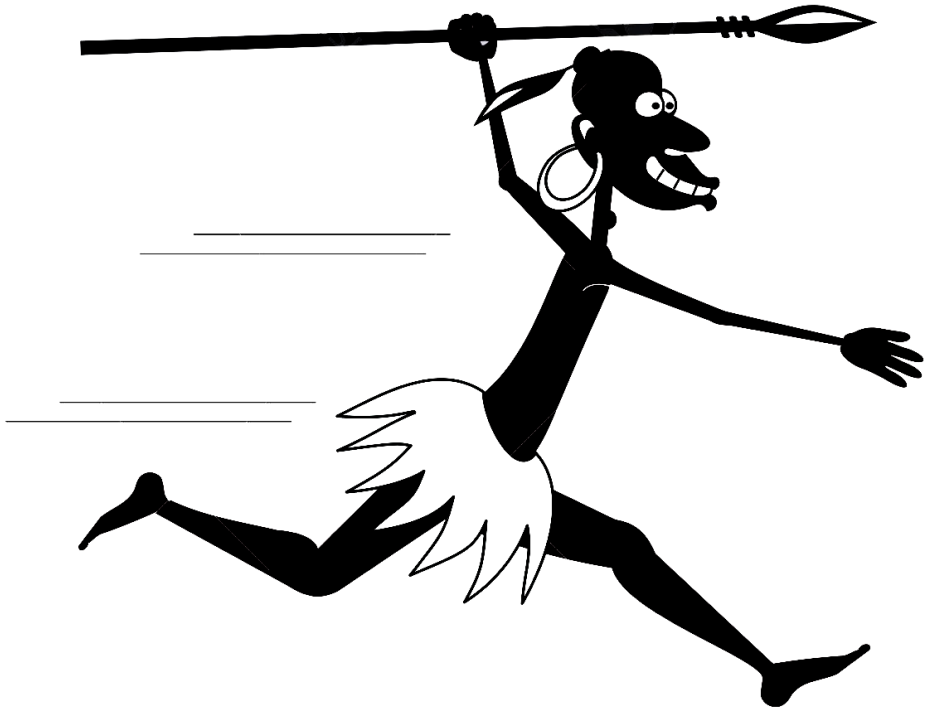
IntStream

LongStream

DoubleStream

`Stream<T>`		
`IntStream`		`Stream<Integer>`
`LongStream`		`Stream<Long>`
`DoubleStream`		`Stream<Double>`

Primitive is faster!



```
int[] ints;
Integer[] integers;

@Setup
public void setup() {
  ints = new Random(1).ints(1000000, 0, 1000)
                      .toArray();

  integers = new Random(1).ints(1000000, 0, 1000)
                          .boxed().toArray(Integer[]::new);
}
```

```
@Benchmark
public long stream() {
  return Stream.of(integers).distinct().count();
}

@Benchmark
public long intStream() {
  return IntStream.of(ints).distinct().count();
}
```

```java
@Benchmark
public long stream() {
  return Stream.of(integers).distinct().count();
}

@Benchmark
public long intStream() {
  return IntStream.of(ints).distinct().count();
}
```

```
# VM version: JDK 1.8.0_92, VM 25.92-b14
Benchmark   Mode  Cnt    Score    Error  Units
intStream   avgt   30  17.121 ± 0.296  ms/op
stream      avgt   30  15.764 ± 0.116  ms/op
```

```
package java.util.stream;
```

```
Stream<T>     ⇒ ReferencePipeline
IntStream     ⇒ IntPipeline
LongStream    ⇒ LongPipeline
DoubleStream  ⇒ DoublePipeline
```

```
// java.util.stream.IntPipeline

@Override
public final IntStream distinct() {
    // While functional and quick to implement,
    // this approach is not very efficient.
    // An efficient version requires an
    // int-specific map/set implementation.
    return boxed().distinct().mapToInt(i -> i);
}
```

```
@Benchmark
public long stream() {
  return Stream.of(integers).distinct().count();
}

@Benchmark
public long intStream() {
  // return IntStream.of(ints).distinct().count();
  return IntStream.of(ints).boxed().distinct()
                           .mapToInt(i -> i).count();
}

-prof gc
stream          48870 B/op
intStream    13642536 B/op
```

Причуда №2

```
private int[] data;

@Setup
public void setup() {
  data = new Random(1).ints(1_000_000, 0, 50_000)
                      .sorted().toArray();
}
```

```
@Benchmark
public int distinct() {
  return IntStream.of(data).distinct().sum();
}
```

```
@Benchmark
public int distinct() {
  return IntStream.of(data).distinct().sum();
}

@Benchmark
public int sortedDistinct() {
  return IntStream.of(data).sorted().distinct().sum();
}

@Benchmark
public int boxedSortedDistinct() {
  return IntStream.of(data).boxed().sorted().distinct()
                           .mapToInt(x -> x).sum();
}
```

```
@Benchmark
public int distinct() {
  return IntStream.of(data).boxed().distinct()
                           .mapToInt(x -> x).sum();
}

@Benchmark
public int sortedDistinct() {
  return IntStream.of(data).sorted().boxed().distinct()
                           .mapToInt(x -> x).sum();
}

@Benchmark
public int boxedSortedDistinct() {
  return IntStream.of(data).boxed().sorted().distinct()
                           .mapToInt(x -> x).sum();
}
```

```
# VM version: JDK 1.8.0_101, VM 25.101-b13
Benchmark              Mode  Cnt   Score   Error  Units
distinct               avgt  100  14.744 ± 1.075  ms/op
sortedDistinct         avgt  100  26.190 ± 0.809  ms/op
boxedSortedDistinct    avgt  100   8.102 ± 0.347  ms/op
```

```
@Benchmark
public int distinct() {
  return IntStream.of(data).boxed().distinct()
                           .mapToInt(x -> x).sum();
}

@Benchmark
public int sortedDistinct() {
  return IntStream.of(data).sorted().boxed().distinct()
                           .mapToInt(x -> x).sum();
}

@Benchmark
public int boxedSortedDistinct() {
  return IntStream.of(data).boxed().sorted().distinct()
                           .mapToInt(x -> x).sum();
}
```

```java
// java.util.stream.IntPipeline

@Override
public final Stream<Integer> boxed() {
    return mapToObj(Integer::valueOf);
}
```

```
@Benchmark
public int distinct() {
  return IntStream.of(data).mapToObj(Integer::valueOf)
                .distinct().mapToInt(x -> x).sum();
}

@Benchmark
public int sortedDistinct() {
  return IntStream.of(data).sorted().mapToObj(Integer::valueOf)
                .distinct().mapToInt(x -> x).sum();
}

@Benchmark
public int boxedSortedDistinct() {
  return IntStream.of(data).mapToObj(Integer::valueOf).sorted()
                .distinct().mapToInt(x -> x).sum();
}
```

JDK-8153293 Preserve SORTED and DISTINCT characteristics for boxed() and asLongStream() operations

```
# VM version: JDK 1.8.0_101, VM 25.101-b13
Benchmark                Mode  Cnt   Score   Error  Units
distinct                 avgt  100  14.744 ± 1.075  ms/op
sortedDistinct           avgt  100  26.190 ± 0.809  ms/op
boxedSortedDistinct      avgt  100   8.102 ± 0.347  ms/op


# VM version: JDK 9-ea, VM 9-ea+139
# VM options: -XX:+UseParallelGC
Benchmark                Mode  Cnt   Score   Error  Units
distinct                 avgt  100   9.800 ± 0.281  ms/op
sortedDistinct           avgt  100   6.573 ± 0.225  ms/op
boxedSortedDistinct      avgt  100   8.777 ± 0.454  ms/op
```

```
# VM version: JDK 9-ea, VM 9-ea+139
# VM options: -XX:+UseParallelGC
Benchmark                Mode  Cnt   Score    Error  Units
distinct                 avgt  100   9.800 ± 0.281  ms/op
sortedDistinct           avgt  100   6.573 ± 0.225  ms/op
boxedSortedDistinct      avgt  100   8.777 ± 0.454  ms/op


# VM version: JDK 9-ea, VM 9-ea+139 (G1)
Benchmark                Mode  Cnt   Score    Error  Units
distinct                 avgt  100   9.377 ± 0.090  ms/op
sortedDistinct           avgt  100   6.360 ± 0.057  ms/op
boxedSortedDistinct      avgt  100  33.924 ± 1.567  ms/op
```

Причуда №3

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

Image courtesy: http://magenta-stock.deviantart.com/

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
// IntStream ints(long streamSize, int origin, int bound)
new Random().ints(5, 1, 20+1).sum();
```

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
// IntStream ints(long streamSize, int origin, int bound)
new Random().ints(5, 1, 20+1).sum();
```

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
// IntStream ints(long streamSize, int origin, int bound)
new Random().ints(5, 1, 20+1).sum();

new Random().ints(5, 1, 20+1).distinct().sum();
```

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
// IntStream ints(long streamSize, int origin, int bound)
new Random().ints(5, 1, 20+1).sum();

new Random().ints(5, 1, 20+1).distinct().sum();

// IntStream ints(int origin, int bound)
new Random().ints(1, 20+1).distinct().limit(5).sum();
```

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
new Random().ints(1, 20+1).distinct().limit(5).sum();

new Random().ints(1, 20+1).parallel().distinct().limit(5).sum();

new Random().ints(1, 20+1).distinct().parallel().limit(5).sum();

new Random().ints(1, 20+1).distinct().limit(5).parallel().sum();
```

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
new Random().ints(1, 20+1).distinct().limit(5).sum();

new Random().ints(1, 20+1).parallel().distinct().limit(5).sum();

new Random().ints(1, 20+1).distinct().parallel().limit(5).sum();

new Random().ints(1, 20+1).distinct().limit(5).parallel().sum();

new Random().ints(1, 20+1).parallel().distinct().limit(5)
            .sequential().sum();
```

JDK-8132800 clarify stream package documentation regarding sequential vs parallel modes

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
new Random().ints(1, 20+1).distinct().limit(5).sum();
```

```
# VM version: JDK 1.8.0_92, VM 25.92-b14
Benchmark  Mode  Cnt       Score      Error  Units
rndSum     avgt   30       0.286 ±    0.001  us/op
```

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
new Random().ints(1, 20+1).distinct().limit(5).sum();
```

```
# VM version: JDK 1.8.0_92, VM 25.92-b14
Benchmark   Mode  Cnt        Score      Error  Units
rndSum      avgt   30        0.286 ±    0.001  us/op
rndSumPar                    ~6080             years/op
```

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
new Random().ints(1, 20+1).distinct().limit(5).sum();
```

```
# VM version: JDK 1.8.0_92, VM 25.92-b14
Benchmark  Mode  Cnt       Score      Error  Units
rndSum     avgt   30       0.286 ±    0.001  us/op
rndSumPar                  ~6080             years/op
```

java.util.stream.StreamSpliterators.UnorderedSliceSpliterator<T, T_SPLITR>

Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

```
public IntStream ints(int randomNumberOrigin,
                      int randomNumberBound)
```

Returns an **effectively** unlimited stream of pseudorandom int values, each conforming to the given origin (inclusive) and bound (exclusive).

Implementation Note:
This method is implemented to be equivalent to `ints(Long.MAX_VALUE, randomNumberOrigin, randomNumberBound)`.

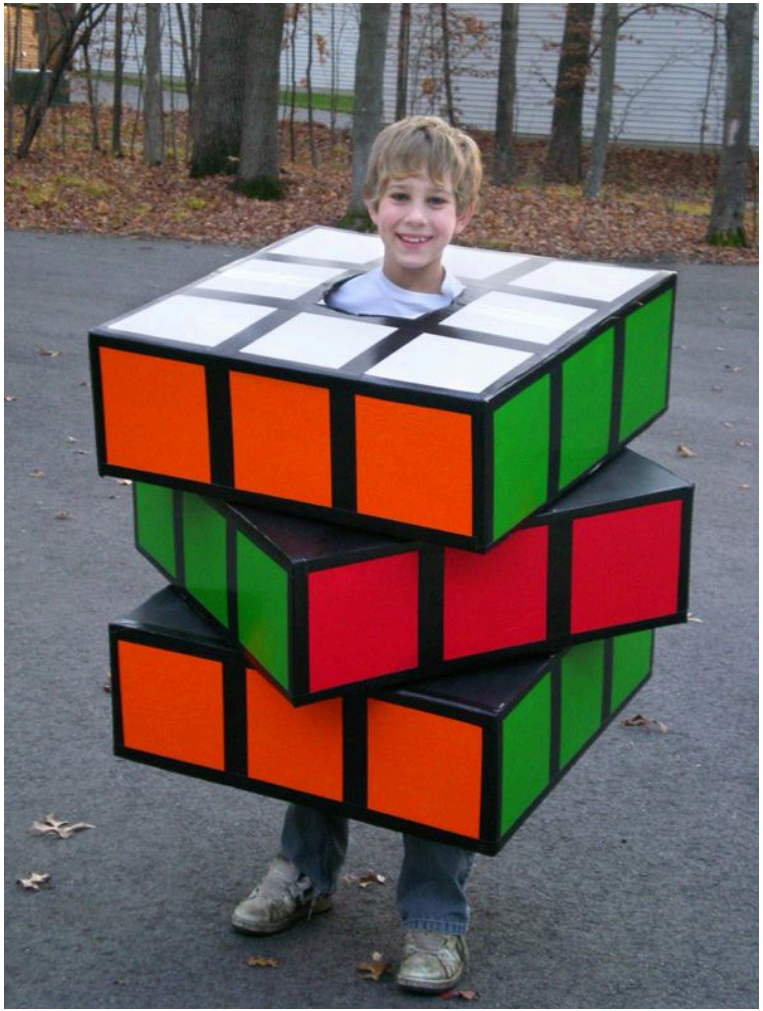
Задача: вычислить сумму 5 различных псевдослучайных чисел от 1 до 20.

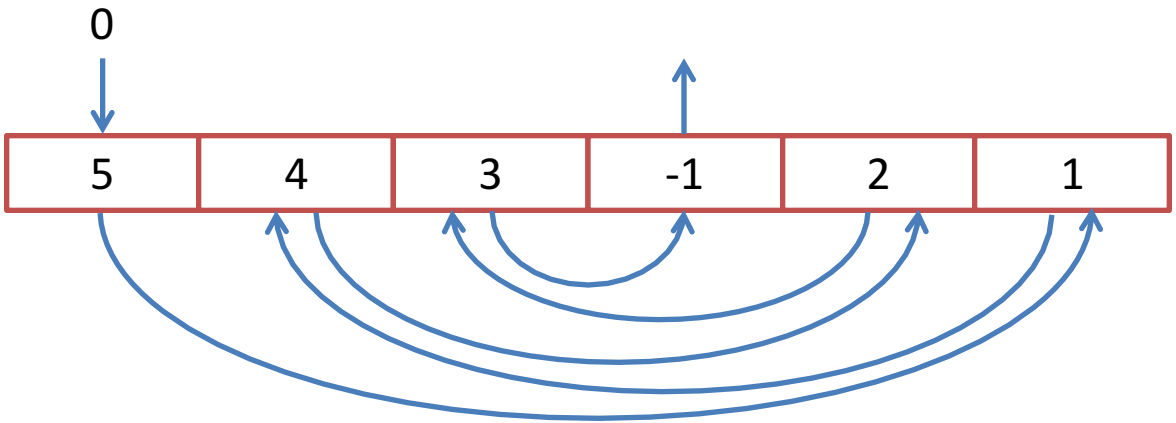
JDK-8154387 Parallel unordered Stream.limit() tries to collect 128 elements even if limit is less

```
new Random().ints(1, 20+1).distinct().limit(5).sum();
new Random().ints(1, 20+1).parallel().distinct().limit(5).sum();
```

```
# VM version: JDK 9-ea, VM 9-ea+130
Benchmark   Mode  Cnt  Score    Error  Units
rndSum      avgt   30  0.318 ± 0.003  us/op
rndSumPar   avgt   30  7.793 ± 0.026  us/op
```

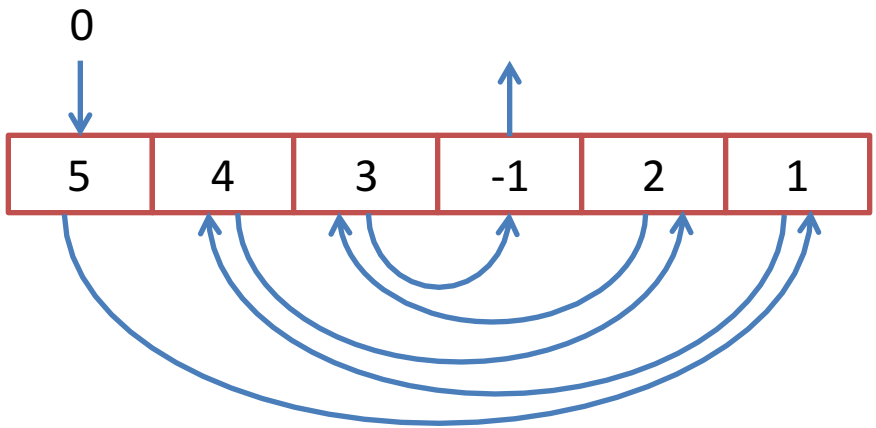
Причуда №4



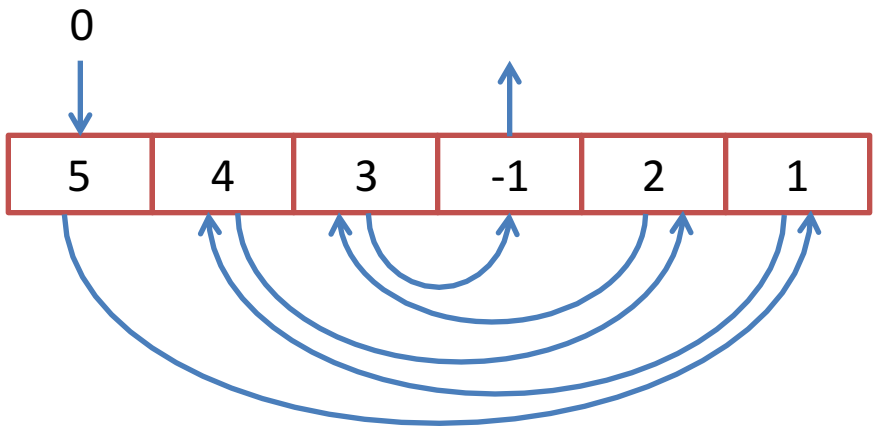

0
5
4
3
-1
2
1

```
int[] perm = { 5, 4, 3, -1, 2, 1 };
Stream.iterate(0, i -> perm[i])
   .skip(1)
   .limit(perm.length)
   .forEach(System.out::println);
```

```
5
1
4
2
3
-1
```

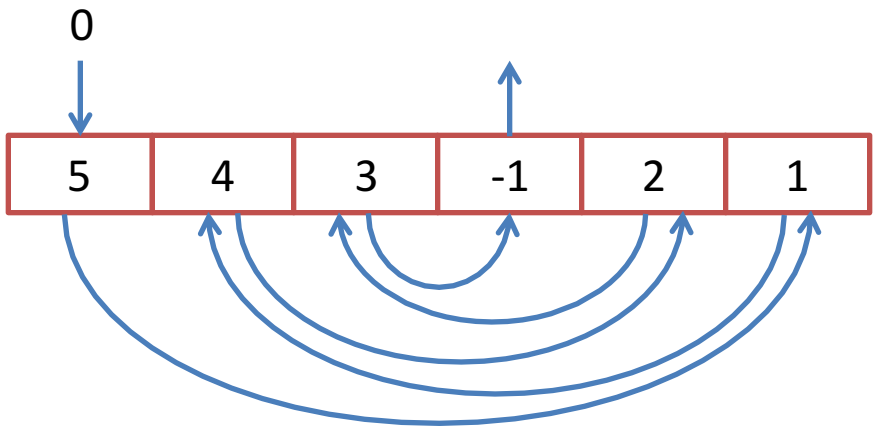


```
int[] perm = { 5, 4, 3, -1, 2, 1 };
IntStream.iterate(0, i -> perm[i])
   .skip(1)
   .limit(perm.length)
   .forEach(System.out::println);
```




```
5
1
4
2
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
    at test.Iterate.lambda$0(Iterate.java:9)
    at test.Iterate$$Lambda$1/424058530.applyAsInt(Unknown Source)
    at java.util.stream.IntStream$1.nextInt(IntStream.java:754)
    ...
    at test.Iterate.main(Iterate.java:12)
```

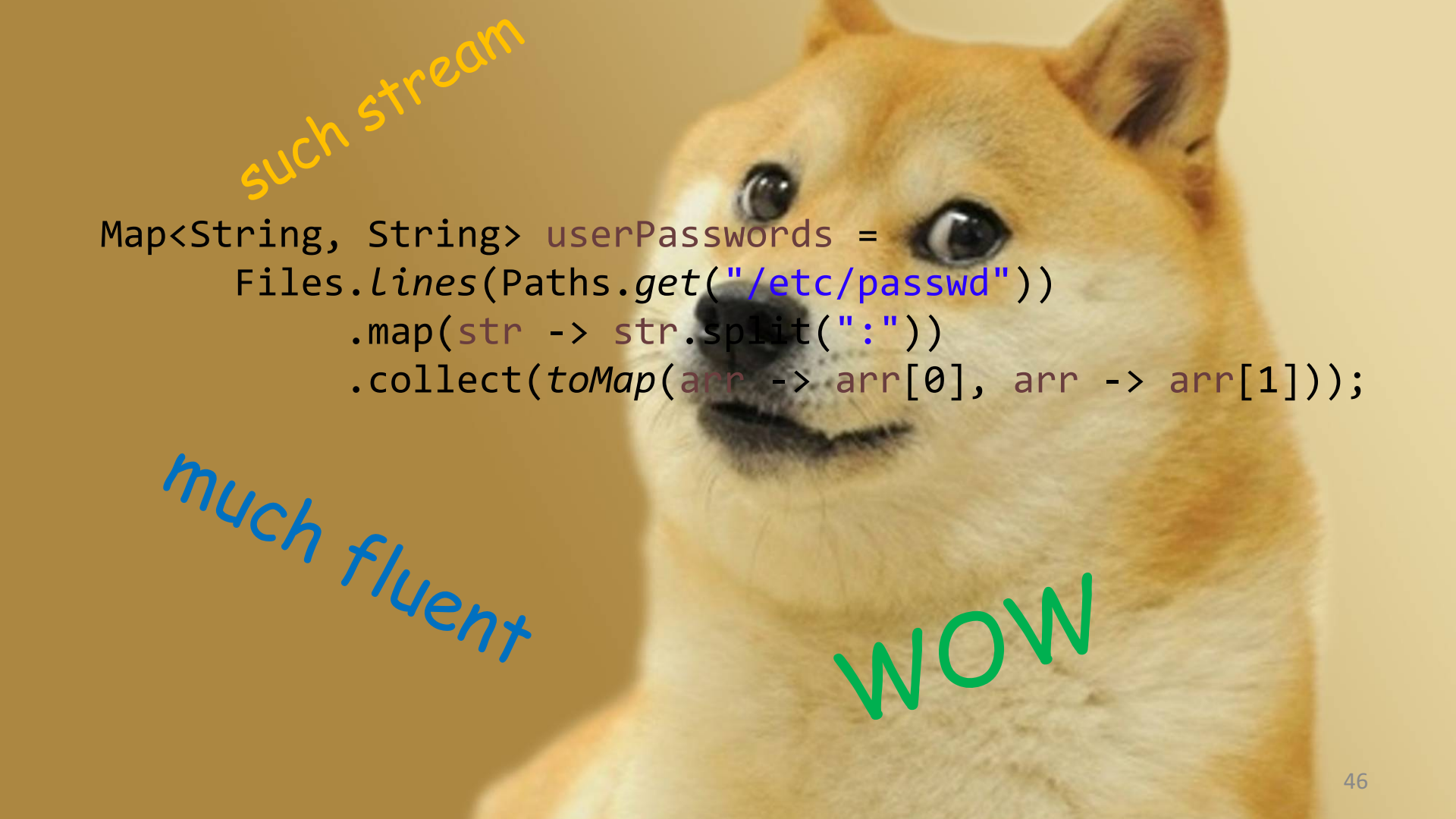
```
int[] perm = { 5, 4, 3, -1, 2, 1 };
IntStream.iterate(0, i -> perm[i])
   .skip(1)
   .limit(perm.length)
   .forEach(System.out::println);
// Java-9
5
1
4
2
3
-1
```



JDK-8072727 add variation of Stream.iterate() that's finite

Причуда №5

```
Map<String, String> userPasswords =
      Files.lines(Paths.get("/etc/passwd"))
           .map(str -> str.split(":"))
           .collect(toMap(arr -> arr[0], arr -> arr[1]));
```




```
Map<String, String> userPasswords =
      Files.lines(Paths.get("/etc/passwd"))
           .map(str -> str.split(":"))
           .collect(toMap(arr -> arr[0], arr -> arr[1]));
```

```
Map<String, String> userPasswords;
try (Stream<String> stream =
      Files.lines(Paths.get("/etc/passwd"))) {
    userPasswords = stream.map(str -> str.split(":"))
        .collect(toMap(arr -> arr[0], arr -> arr[1]));
}
```

```
Map<String, String> userPasswords;
try (Stream<String> stream =
      Files.lines(Paths.get("/etc/passwd"))) {
    userPasswords = stream.map(str -> str.split(":"))
        .collect(toMap(arr -> arr[0], arr -> arr[1]));
}
```

http://stackoverflow.com/questions/34753078/

16

1

Using the Java 8 `Stream` API, I would like to register a "completion hook", along the lines of:

```
Stream<String> stream = Stream.of("a", "b", "c");

// additional filters / mappings that I don't control
stream.onComplete((Completion c) -> {
    // This is what I'd like to do:
    closeResources();

    // This might also be useful:
    Optional<Throwable> exception = c.exception();
    exception.ifPresent(e -> throw new ExceptionWrapper(e));
});
```

The reason why I want to do that is because I want to wrap a resource in a `Stream` for API clients to consume, and I want that `Stream` to clean up the resource automatically once it is consumed. If that were possible, then the client could call:

```
Collected collectedInOneGo =
Utility.something()
       .niceLookingSQLDSL()
       .moreDSLFeatures()
       .stream()
       .filter(a -> true)
       .map(c -> c)
```

Терминальные операции

Нормальные
(с внутренним обходом):

```
forEach()
collect()
reduce()
findFirst()
count()
toArray()
…
```

Терминальные операции

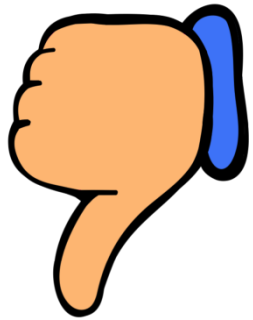
Нормальные
(с внутренним обходом):

```
forEach()
collect()
reduce()
findFirst()
count()
toArray()
```

…

Причудливые
(с внешним обходом):

```
iterator()
spliterator()
```



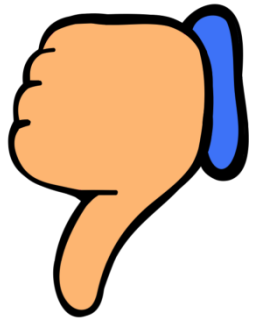
Терминальные операции

Нормальные
(с внутренним обходом):

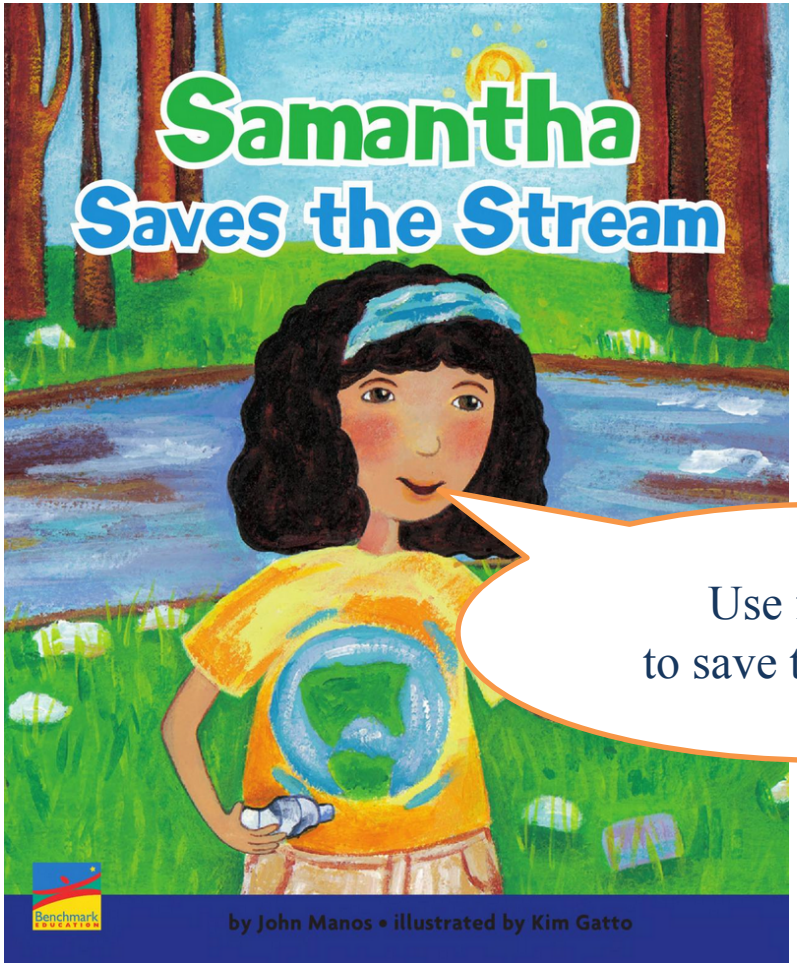
```
forEach()
collect()
reduce()
findFirst()
count()
```

Причудливые
(с внешним обходом):

```
iterator()
spliterator()
```



Stream.concat(a, b).findFirst() → a.spliterator().tryAdvance(…);



Use flatMap
to save the Stream!

flatMap

<R> Stream<R> flatMap(
 Function<? super T,? extends Stream<? extends R>> mapper)

Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

```
Map<String, String> userPasswords =
    Stream.of(Files.lines(Paths.get("/etc/passwd")))
          .flatMap(s -> s)
          .map(str -> str.split(":"))
          .collect(toMap(arr -> arr[0], arr -> arr[1]));
```

flatMap() vs concat()

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

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);
```

```
Files.lines(Paths.get("/etc/passwd"))
          .spliterator().tryAdvance(...);
// вычитываем одну строку, файл не закрываем

Stream.of(Files.lines(Paths.get("/etc/passwd")))
          .flatMap(s -> s)
          .spliterator().tryAdvance(...);
// вычитываем весь файл в память, файл закрываем
```

```
Files.lines(Paths.get("/etc/passwd"))
          .spliterator().tryAdvance(...);
// вычитываем одну строку, файл не закрываем

Stream.of(Files.lines(Paths.get("/etc/passwd")))
          .flatMap(s -> s)
          .spliterator().tryAdvance(...);
// вычитываем весь файл в память, файл закрываем
```

http://stackoverflow.com/a/32767282/4856258

(небуферизующий flatMap)

There’s no way to save the Stream ☹

Причуда №6

Задача №1: посчитать количество 1 000 000 среди чисел от 0 до 1 000 000
Задача №2: найти число 1 000 000 среди чисел от 0 до 1 000 000

```
@Benchmark
public long count() {
  return IntStream.rangeClosed(0, 1_000_000).boxed()
          .filter(x -> x == 1_000_000).count();
}

@Benchmark
public Optional<Integer> findAny() {
  return IntStream.rangeClosed(0, 1_000_000).boxed()
          .filter(x -> x == 1_000_000).findAny();
}
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                  Mode  Cnt  Score   Error  Units
FindAnyTest.count          avgt   30  4.057 ± 0.025  ms/op
FindAnyTest.findAny        avgt   30  5.098 ± 0.021  ms/op (1.25×)
```

```
List<Integer> list;

@Setup
public void setup() {
  list = IntStream.rangeClosed(0, 1_000_000).boxed()
           .collect(Collectors.toCollection(ArrayList::new));
}

@Benchmark
public long count() {
  return list.stream().filter(x -> x == 1_000_000).count();
}

@Benchmark
public Optional<Integer> findAny() {
  return list.stream().filter(x -> x == 1_000_000).findAny();
}
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                  Mode  Cnt  Score   Error  Units
FindAnyTest.count          avgt   30  4.057 ± 0.025  ms/op
FindAnyTest.findAny        avgt   30  5.098 ± 0.021  ms/op (1.25×)

ListTest.count             avgt   30  2.507 ± 0.029  ms/op
ListTest.findAny           avgt   30  4.142 ± 0.043  ms/op (1.65×)
```

```
@Benchmark
public Optional<Integer> findAny1() {
  return IntStream.range(0, 1_000_000)
                  .boxed().filter(x -> x == 0).findAny();
}

@Benchmark
public Optional<Integer> findAny1Flat() {
  return IntStream.of(1_000_000).flatMap(x -> IntStream.range(0, x))
                  .boxed().filter(x -> x == 0).findAny();
}
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                    Mode  Cnt      Score     Error  Units
FindAnyTest.findAny1         avgt   30      0.083 ±   0.001  us/op
FindAnyTest.findAny1Flat     avgt   30   4486.370 ±  37.959  us/op
                                         (54000×)
```

Короткозамкнутые операции:

- Медленнее, чем обычные (tryAdvance)
- Не останавливаются внутри стрима, созданного flatMap


Шурик, Вы комсомолец?
Это же не наш метод! Где гуманизм?

http://c2.com/cgi/wiki?DontUseExceptionsForFlowControl

Dont Use Exceptions For Flow Control

[ExceptionPatterns]

Exceptions (in Java and C++) are like non-local goto statements. As such they can be used to build general control flow constructs. For example, you could write a recursive search for a binary tree which used an exception to return the result:

```
void search( TreeNode node, Object data ) throws ResultException {
if (node.data.equals( data ))
        throw new ResultException( node );
else {
        search( node.leftChild, data );
        search( node.rightChild, data );
}
}
```

The cute trick here is that the exception will break out of the recursion in one step no matter how deep it has g… (…is is a linear search rather than a binary one, despite the binary tree. Better example requested.)

This violates the PrincipleOfLeastAstonishment. This makes it harder for programmers to read. There … exis… control structures whose sole purpose is to handle these types of operations. Be kind to the develope… who follo… you and use a standard approach rather than being creative.

Perhaps more important, it's not what compiler implementors expect. They exp… e… …e set up often but thrown rarely, and they usually let the throw code be quite inefficient. Throwing ex…ti… on… of the most expensive operations in Java, surpassing even *new*. On the other hand, don't forget the first … of …timi…ation (RulesOfOptimization).

Another example might be code like:

```
try {
for (int i = 0; /*wot no test?*/ ;…)
        array[i]++;
} catch (ArrayIndexOutOfBoundsExcep… …) {}
```

This is equivalent to:

```
for (int i = 0; i < array.length; i++)
```

ANTIPATTERN

— Надо, Федя. Надо!

```
static class FoundException extends RuntimeException {
  Object payload;

  FoundException(Object payload) { this.payload = payload; }
}

public static <T> Optional<T> fastFindAny(Stream<T> stream) {
  try {
    stream.forEach(x -> { throw new FoundException(x); });
    return Optional.empty();
  } catch (FoundException fe) {
    @SuppressWarnings({ "unchecked" })
    T val = (T) fe.payload;
    return Optional.of(val);
  }
}
```

```
@Benchmark
public Optional<Integer> findAny() {
  return IntStream.rangeClosed(0, 1_000_000).boxed()
        .filter(x -> x == 1_000_000).findAny();
}

@Benchmark
public Optional<Integer> fastFindAny() {
  return fastFindAny(IntStream.rangeClosed(0, 1_000_000)
        .boxed().filter(x -> x == 1_000_000));
}
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                    Mode  Cnt  Score   Error  Units
FindAnyTest.count            avgt   30  4.057 ± 0.025  ms/op
FindAnyTest.findAny          avgt   30  5.098 ± 0.021  ms/op (1.25×)
FindAnyTest.fastFindAny      avgt   30  4.111 ± 0.083  ms/op

ListTest.count               avgt   30  2.507 ± 0.029  ms/op
ListTest.findAny             avgt   30  4.142 ± 0.043  ms/op (1.65×)
ListTest.fastFindAny         avgt   30  2.540 ± 0.059  ms/op
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                          Mode  Cnt      Score     Error  Units
FindAnyTest.findAny1               avgt   30      0.083 ±   0.001  us/op
FindAnyTest.findAny1Flat           avgt   30   4486.370 ±  37.959  us/op

FindAnyTest.fastFindAny1           avgt   30      1.785 ±   0.008  us/op
FindAnyTest.fastFindAny1Flat       avgt   30      2.012 ±   0.012  us/op
```

```
protected RuntimeException(String message,
                           Throwable cause,
                           boolean enableSuppression,
                           boolean writableStackTrace)
```

Constructs a new runtime exception with the specified detail message, cause, suppression enabled or disabled, and writable stack trace enabled or disabled.

Parameters:

`message` - the detail message.

`cause` - the cause. (A `null` value is permitted, and indicates that the cause is nonexistent or unknown.)

`enableSuppression` - whether or not suppression is enabled or disabled

`writableStackTrace` - whether or not the stack trace should be writable

Since:

`1.7`

```
static class FoundException extends RuntimeException {
  Object payload;

  FoundException(Object payload) {
    super("", null, false, false); // <<<<
    this.payload = payload;
  }
}
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                              Mode  Cnt      Score     Error  Units
FindAnyTest.findAny1                   avgt   30      0.083 ±   0.001  us/op
FindAnyTest.findAny1Flat               avgt   30   4486.370 ± 37.959  us/op

FindAnyTest.fastFindAny1               avgt   30      1.785 ±   0.008  us/op
FindAnyTest.fastFindAny1Flat           avgt   30      2.012 ±   0.012  us/op

// no stack trace
FindAnyTest.fastFindAny1               avgt   30      0.280 ±   0.016  us/op
FindAnyTest.fastFindAny1Flat           avgt   30      0.342 ±   0.021  us/op
```

```
public static <T> Optional<T> fastFindAny(Stream<T> stream) {
  try {
    stream.forEach(x -> { throw new FoundException(x); });
    return Optional.empty();
  } catch (FoundException fe) {
    @SuppressWarnings({ "unchecked" })
    T val = (T) fe.payload;
    return Optional.of(val);
  }
}
```

```
@Benchmark
public Optional<Integer> findAnyPar() {
    return IntStream.range(0, 100_000_000).parallel().boxed()
            .filter(x -> x == 10_000_000).findAny();
}

@Benchmark
public Optional<Integer> fastFindAnyPar() {
    return fastFindAny(IntStream.range(0, 100_000_000)
         .parallel().boxed().filter(x -> x == 10_000_000));
}
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                          Mode  Cnt      Score     Error  Units
FindAnyTest.findAnyPar             avgt   30     39.112 ±   0.742  ms/op
FindAnyTest.fastFindAnyPar         avgt   30     61.896 ±   4.329  ms/op
```

Image courtesy: http://www.scalingbits.com/java/javakurs2/programmieren2/threads

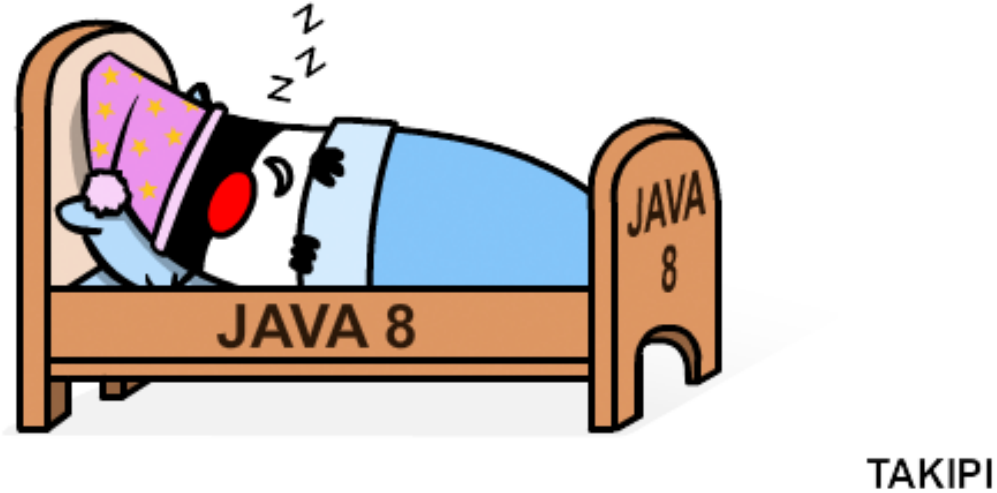


Image courtesy: http://blog.takipi.com/while-you-were-sleeping-the-top-new-java-8-additions/

```
@Param({ "0", "20", "40" })
private int sleep;

@Setup(Level.Invocation)
public void setup() throws InterruptedException {
    Thread.sleep(sleep);
}
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                (sleep)  Mode  Cnt     Score     Error  Units
findAnyPar                        avgt   30    39.112 ±   0.742  ms/op
fastFindAnyPar                0   avgt   30    61.896 ±   4.329  ms/op
fastFindAnyPar               20   avgt   30    47.033 ±   3.054  ms/op
fastFindAnyPar               40   avgt   30    42.050 ±   0.520  ms/op
```

```
AtomicInteger i = new AtomicInteger();
Optional<Integer> result = fastFindAny(
    IntStream.range(0, 100_000_000).parallel()
             .boxed().peek(e -> i.incrementAndGet())
             .filter(x -> x == 10_000_000));
System.out.println(i);
System.out.println(result);
Thread.sleep(1000);
System.out.println(i);
```

```
AtomicInteger i = new AtomicInteger();
Optional<Integer> result = fastFindAny(
    IntStream.range(0, 100_000_000).parallel()
             .boxed().peek(e -> i.incrementAndGet())
             .filter(x -> x == 10_000_000));
System.out.println(i);
System.out.println(result);
Thread.sleep(1000);
System.out.println(i);
>> 20836268
>> Optional[10000000]
>> 52620467
```

```
# VM version: JDK 1.8.0_91, VM 25.91-b14
Benchmark                (sleep)  Mode  Cnt     Score     Error  Units
findAnyPar                        avgt   30    39.112 ±   0.742  ms/op
fastFindAnyPar                0   avgt   30    61.896 ±   4.329  ms/op
fastFindAnyPar               20   avgt   30    47.033 ±   3.054  ms/op
fastFindAnyPar               40   avgt   30    42.050 ±   0.520  ms/op
```

JDK-8164690 Exceptions from a Fork/Join task to serve as an implicit cancellation

Спасибо за внимание

https://twitter.com/tagir_valeev

https://github.com/amaembo

https://habrahabr.ru/users/lany