ORACLE

JDK 8: Stream Style

Сергей Куксенко
sergey.kuksenko@oracle.com, @kuksenk0

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Мотивация

Мотивация

Зачем нам какие-то Stream'ы, когда и так всё хорошо?

Мотивация

Зачем нам какие-то Stream'ы, когда и так всё хорошо?

```
public void printGroups(List<People> people) {
    Set<Group> groups = new HashSet<>();
    for (Person p : people) {
        if (p.getAge() >= 65)
            groups.add(p.getGroup());
    }
    List<Group> sorted = new ArrayList<>(groups);
    Collections.sort(sorted, new Comparator<Group>() {
        public int compare(Group a, Group b) {
            return Integer.compare(a.getSize(), b.getSize())
        }
    });
    for (Group g : sorted)
        System.out.println(g.getName());
}
```

Мотивация

Было бы круто не городить километры одинакового кода:

```
public void printGroups(List<People> people) {
    people.stream()
        .filter(p -> p.getAge() > 65)
        .map(p -> p.getGroup())
        .distinct()
        .sorted(comparing(g -> g.getSize())
        .map(g -> g.getName())
        .forEach(n -> System.out.println(n));
}
```

Мотивация

Laziness:

Было бы круто работать попозже (поменьше):

```
public void printGroups(List<People> people) {
    people.stream()
        .filter(p -> p.getAge() > 65)          // спим
        .map(p -> p.getGroup())                // тимбилдинг
        .distinct()                            // покупаем iPhone
        .sorted(comparing(g -> g.getSize()))   // читаем почту
        .map(g -> g.getName())                 // спорим
        .forEach(n -> System.out.println(n));⇐ ДЕДЛАЙН!!!
}
```

Мотивация

Параллелизм?

```
Collection<Item> data;
...


for(int i=0; i < data.size(); i++) {
    processItem(data.get(i));
}
```

Мотивация

Параллелизм?

```
Collection<Item> data;
...


for(Item item : data) {
    processItem(item);
}
```

Мотивация

Параллелизм?

```
Collection<Item> data;
...

#pragma ~~omg~~omp parallel
for(Item item : data) {
    processItem(item);
}
```

Мотивация

Параллелизм?

```
Collection<Item> data;
...


parallel_for(Item item : data) {
    processItem(item);
}
```

Мотивация

Параллелизм!

```
Collection<Item> data;
...


data.parallelStream()
    .forEach(item -> processItem(item));
```

Дизайн

Дизайн

- Большинство кода укладывается в простой паттерн:

$source$

Дизайн

- Большинство кода укладывается в простой паттерн:

$$source \rightarrow op$$

Дизайн

- Большинство кода укладывается в простой паттерн:

$$source \rightarrow op \rightarrow op$$

Дизайн

- Большинство кода укладывается в простой паттерн:

$$source \rightarrow op \rightarrow op \rightarrow op$$

Дизайн

- Большинство кода укладывается в простой паттерн:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow$$

Дизайн

- Большинство кода укладывается в простой паттерн:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow gangnamstyle$$

Дизайн

- Большинство кода укладывается в простой паттерн:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow sink$$

Дизайн

- Большинство кода укладывается в простой паттерн:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow sink$$

- «sources»: collections, iterators, channels, ...
- «operations»: filter, map, reduce, ...
- «sinks»: collections, locals, ...

sources

- стандартные классы как источники?
  - + классы, которые ещё не написаны
  - + классы, написанные пользователями

sources

- стандартные классы как источники?
  - + классы, которые ещё не написаны
  - + классы, написанные пользователями
- `Collection` не подходит
  - не всё же сначала в коллекцию заворачивать?

sources

- стандартные классы как источники?
  - + классы, которые ещё не написаны
  - + классы, написанные пользователями
- `Collection` не подходит
  - не всё же сначала в коллекцию заворачивать?
- `Iterable` не подходит
  - адовы проблемы с последовательными итераторами
  - методы для операций пачкают интерфейс

sources

- стандартные классы как источники?
  - + классы, которые ещё не написаны
  - + классы, написанные пользователями
- `Collection` не подходит
  - не всё же сначала в коллекцию заворачивать?
- `Iterable` не подходит
  - адовы проблемы с последовательными итераторами
  - методы для операций пачкают интерфейс
- `Stream` подходит
  - отдельная штука с нужной нам семантикой
  - «пачкаем» классы только методом `stream()`

Stream

Stream

- Stream = «a multiplicity of values»
- элементы не упорядочены
- не структура данных (no storage)
- выполнение операций отложено до последнего
- может быть бесконечным
- не мутирует источник
- одноразовый
- позволяет как последовательную, так и паралелльную обработку
- есть примитивные специализации: `IntStream`, `LongStream`, `DoubleStream`

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
    people.stream()
        .filter(p -> p.getAge() > 65)
        .map(p -> p.getGroup())
        .distinct()
        .sorted(comparing(g -> g.getSize())
        .map(g -> g.getName())
        .forEach(n -> System.out.println(n));
}
```

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
    people.stream()
        .filter(p -> p.getAge() > 65)
        .map(p -> p.getGroup())
        .distinct()
        .sorted(comparing(g -> g.getSize())
        .map(g -> g.getName())
        .forEach(n -> System.out.println(n));
}
```

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
    people.stream()
        .filter(p -> p.getAge() > 65)
        .map(p -> p.getGroup())
        .distinct()
        .sorted(comparing(g -> g.getSize())
        .map(g -> g.getName())
        .forEach(n -> System.out.println(n));
}
```

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
    Stream<People> s1 = people.stream();
    Stream<People> s2 = s1.filter(p -> p.getAge() > 65);
    Stream<Group>  s3 = s2.map(p -> p.getGroup());
    Stream<Group>  s4 = s3.distinct();
    Stream<Group>  s5 = s4.sorted(comparing(g -> g.getSize()));
    Stream<String> s6 = s5.map(g -> g.getName());
    s6.forEach(n -> System.out.println(n));
}
```

Характеристки Stream'a[1]

ordered

distinst

sorted

sized

notnull

immutable

concurrent

subsized

...

[1] обычно не видны и не нужны снаружи

Stream Sources

Stream Sources: Коллекции

```
ArrayList<T> list;
Stream<T> s = list.stream();
// sized, ordered
```

Stream Sources: Коллекции

```
ArrayList<T> list;
Stream<T> s = list.stream();
// sized, ordered
```

```
HashSet<T> set;
Stream<T> s = set.stream();
// sized, distinct
```

```
TreeSet<T> set;
Stream<T> s = set.stream();
// sized, distinct, ordered, sorted
```

Stream Sources: Утилиты

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

Stream Sources: Утилиты

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

```
Stream<T> s = Stream.of(v0, v1, v2);
```

Stream Sources: Утилиты

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

```
Stream<T> s = Stream.of(v0, v1, v2);
```

```
Stream<T> s = Stream.builder()
                    .add(v0).add(v1).add(v2)
                    .build();
```

Stream Sources: Утилиты

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

```
Stream<T> s = Stream.of(v0, v1, v2);
```

```
Stream<T> s = Stream.builder()
                   .add(v0).add(v1).add(v2)
                   .build();
```

```
IntStream s = IntStream.range(0, 100);
```

Stream Sources: Генераторы

```
AtomicInteger init = new AtomicInteger(0);
Stream<Integer> s =
    Stream.generate(init::incrementAndGet);
```

Stream Sources: Генераторы

```
AtomicInteger init = new AtomicInteger(0);
Stream<Integer> s =
    Stream.generate(init::incrementAndGet);
```

```
IntStream s = Stream.iterate(0, i -> i+1);
```

Stream Sources: Прочее

```
Stream<String> s = bufferedReader.lines();
```

Stream Sources: Прочее

```
Stream<String> s = bufferedReader.lines();
```

```
Stream<String> s = Pattern.compile(myRegEx)
                         .splitAsStream(myStr);
```

Stream Sources: Прочее

```
Stream<String> s = bufferedReader.lines();
```

```
Stream<String> s = Pattern.compile(myRegEx)
                           .splitAsStream(myStr);
```

```
DoubleStream s =
            new SplittableRandom().doubles();
```

Intermediate Operations

Intermediate Operations

```
Stream<S> s;
Stream<S> s.filter(Predicate<S>);
Stream<T> s.map(Function<S, T>);
Stream<T> s.flatMap(Function<S, Stream<T>>);
Stream<S> s.peek(Consumer<T>);
Stream<S> s.sorted();
Stream<S> s.distinct();
Stream<S> s.unordered();
Stream<S> s.limit(long);
Stream<S> s.substream(long);
Stream<S> s.substream(long, long);
```

Intermediate Operations

```
Stream<S> s;
Stream<S> s.filter(Predicate<S>);
Stream<T> s.map(Function<S, T>);
Stream<T> s.flatMap(Function<S, Stream<T>>);
Stream<S> s.peek(Consumer<T>);
Stream<S> s.sorted();
Stream<S> s.distinct();
Stream<S> s.unordered();
Stream<S> s.limit(long);
Stream<S> s.substream(long);
Stream<S> s.substream(long, long);

Stream<S> s.parallel();
Stream<S> s.sequential();
```

Terminal Operations a.k.a. PROFIT

Terminal Operations

- терминальные операции дают результат
- параллельно или последовательно (в зависимости от того, что уже есть в наборе операций)

- можно выделить
  - «итераторы»: `forEach`, `iterator`
  - поиск: `findFirst`, `findAny`
  - проверка: `allMatch`, `anyMatch`, `noneMatch`
  - аггрегаторы
    - *reducers*
    - *collectors*

Short-circuting

- некоторые операции могут «бросить» поток
- получают смысл операции над бесконечными потоками

- примеры: `find*`, `*Match`

```
int v = Stream.generate(() -> x++)
                .findFirst().get();
```

«Итераторы»

- делают действие над каждым элементом потока:

```
IntStream.range(0, 100)
    .forEach(System.out::println);
```

- можно вытаскивать элементы из потока по очереди[2]:

```
Iterator<Integer> =
  IntStream.range(0, 100).iterator()[3];
```

---

[2] больше необходимо для совместимости

[3] единственная ленивая терминальная операция

Reducers

Нужна сумма элементов `Stream<Integer>`.
Что делаем?

Reducers

Нужна сумма элементов Stream<Integer>.
Что делаем?

```
public int getSum(Stream<Integer> s){
    int sum;
    s.forEach( i -> sum+=i );
    return sum;
}
```

Reducers

Нужна сумма элементов Stream<Integer>.
Что делаем?

```
public int getSum(Stream<Integer> s){
    int sum;
    s.forEach( i -> sum+=i ); // Compile error
    return sum;
}
```

Reducers

Нужна сумма элементов Stream<Integer>.
Что делаем?

```
public int getSum(Stream<Integer> s){
    int[] sum = new int[1];
    s.forEach( i -> sum[0]+=i );
    return sum[0];
}
```

Reducers

Какой будет результат?

```
IntStream.range(0,100).map(i->1)
```

Reducers

Какой будет результат?

```
IntStream.range(0,100).map(i->1)
```

100

Reducers

Какой будет результат?

```
IntStream.range(0,100).map(i->1)
```

100

```
IntStream.range(0,100).map(i->1).parallel()
```

Reducers

Какой будет результат?

```
IntStream.range(0,100).map(i->1)
```

100

```
IntStream.range(0,100).map(i->1).parallel()
```

79, 63, 100, ...

Reducers

- берут поток и дают некоторый скаляр:

```
int s =
    stream.reduce(0, (x, y) -> x + y);
```

Reducers

- берут поток и дают некоторый скаляр:

```
int s =
    stream.reduce(0, (x, y) -> x + y);
```

- некоторые отдают Optional<T>, чтобы отличать пустоту:

```
Optional<Integer> o =
    stream.reduce((x, y) -> x + y);
Integer i =
    stream.reduce(0, (x, y) -> x + y);
```

Sinks/Collectors

- складывают содержимое потока:

```
List<Integer> list =
    IntStream.range(0, 100).boxed()
        .collect(Collectors.toList());
```

Sinks/Collectors

- складывают содержимое потока:

```
List<Integer> list =
  IntStream.range(0, 100).boxed()
      .collect(Collectors.toList());
```

- могут принимать сложные коллекции:

```
Map<Integer,Integer> map =
  IntStream.range(0, 1000).boxed()
      .collect(
          Collectors.toConcurrentMap(
            (k) -> k % 42,
            Functions.identity(),
            Collectors.lastWinsMerger()
          )
       );
```

Parallelism

parallelism

- многие источники хорошо бьются на части
- многие операции хорошо параллелизуются
- библиотека делает всю работу за нас
- «под капотом» используется `ForkJoinPool`
- но: нужно эксплицитно просить библиотеку

```
int v = list.parallelStream()
                .reduce(Math::max)
                .get();
```

explicit parallelism

Q: Почему не имплицитно?

A: Выигрыш сильно зависит от:

- N – количества элементов в источнике
- Q – стоимости операции над элементом
- P – доступного паралеллизма на машине
- C – количества конкуррентных клиентов

Точно знаем только N.

Неплохо представляем себе P.

Умеем худо-бедно справляться с C.

Q очень сложно оценить.

Splitterator

Demo

Ресурсы

Ресурсы Полезные ссылки

- JDK8 (was Project Lambda):
  http://openjdk.java.net/projects/jdk8/
- Binary builds:
  https://jdk8.java.net/download.html
- Mailing list:
  lambda-dev@openjdk.java.net

Q & A ?

yet another example

```
for(Method m : enclosingCandidate.getDeclaredMethods()){
  if(m.getName().equals(enclosingInfo.getName()) ) {
    Class<?>[] candidateParamClasses = m.getParameterTypes();
    if(candidateParamClasses.length == parameterClasses.length){
      boolean matches = true;
      for(int i = 0; i < candidateParamClasses.length; i++) {
        if(!candidateParamClasses[i].equals(parameterClasses[i])){
            matches = false;
            break;
        }
      }
      if(matches) { // finally, check return type
        if(m.getReturnType().equals(returnType) )
            return m;
      }
    }
  }
}
throw new InternalError("Enclosing method not found");
```

yet another example

```
return Arrays
.stream(enclosingInfo.getEnclosingClass().getDeclaredMethods())
.filter(m->Objects.equals(m.getName(), enclosingInfo.getName())
.filter(m->Arrays.equals(m.getParameterTypes(),parameterClasses))
.filter(m->Objects.equals(m.getReturnType(), returnType))
.findFirst()
.orElseThrow(()->new InternalError("Enclosing␣method␣not␣found");
```