

Stream выражения

Рассматриваемые вопросы

- Понятие Stream
- Иерархия интерфейсов и методы Stream
- Как получить Stream
- Операции со Stream
- Примеры

Понятие Stream



Stream API — API для работы со структурами данных в функциональном стиле.

Stream API — это по своей сути поток данных. Сам термин "поток" довольно размыт в программировании в целом и в Java в частности.

Stream API позволяет писать существенно короче то, что раньше занимало много строк кода, а именно — упростить работу с наборами данных, например, операции фильтрации, сортировки и другие манипуляции с данными.

Понятие Stream

1. “Трубопровод” для обработки данных
2. Содержит последовательность объектов
3. Работает с источниками данных: массивы и коллекции
4. Не хранит данные в себе
5. Это не InputSream/OutputStream!

BaseStream

В основе Stream API лежит интерфейс BaseStream. Его полное определение:

```
interface BaseStream<T , S extends BaseStream<T , S>>
```

Здесь параметр T означает тип данных в потоке, а S - тип потока, который наследуется от интерфейса BaseStream.

BaseStream определяет базовый функционал для работы с потоками, которые реализуются через его методы:

- **void close():** закрывает поток
- **boolean isParallel():** возвращает true, если поток является параллельным
- **Iterator<T> iterator():** возвращает ссылку на итератор потока
- **S parallel():** возвращает параллельный поток (параллельные потоки могут задействовать несколько ядер процессора в многоядерных архитектурах)
- **S sequential():** возвращает последовательный поток
- **S unordered():** возвращает неупорядоченный поток

Иерархия стримов

При работе с потоками, которые представляют определенный примитивный тип - double, int, long проще использовать интерфейсы **DoubleStream**, **IntStream**, **LongStream**.

Но в большинстве случаев, как правило, работа происходит с более сложными данными, для которых предназначен интерфейс `Stream<T>`.

Stream API

Несмотря на то, что все эти операции позволяют взаимодействовать с потоком как неким набором данных наподобие коллекции, важно понимать отличие коллекций от потоков:

- Потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.
- Операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.
- Для потоков характерно отложенное выполнение. То есть выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

Способы создания стримов

| Способ создания стрима | Шаблон создания | Пример |
|--|--|--|
| 1. Классический: Создание стрима из коллекции | <code>collection.stream()</code> <code>collection.parallelStream()</code> | <pre>Collection<String> collection = Arrays.asList("a1", "a2", "a3"); Stream<String> streamFromCollection = collection.stream();</pre> |
| 2. Создание стрима из значений | <code>Stream.of(значение1, ... значениеN)</code> | <pre>Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");</pre> |
| 3. Создание стрима из массива | <code>Arrays.stream(массив)</code> | <pre>String[] array = {"a1", "a2", "a3"}; Stream<String> streamFromArrays = Arrays.stream(array);</pre> |
| 4. Создание стрима из файла (каждая строка в файле будет отдельным элементом в стриме) | <code>Files.lines(путь_к_файлу)</code> | <pre>Stream<String> streamFromFiles = Files.lines(Paths.get("file.txt"))</pre> |
| 5. Создание стрима из строки | «строка».chars() | <pre>IntStream streamFromString = "123".chars()</pre> |

Операции со Stream



Операции (методы) Stream можно разделить на две группы:

1. **промежуточные (intermediate)** – производят другую Stream и используются для создания цепочки действий над объектами

2. **заключительные (terminal)** – побуждают к выполнению промежуточных операция и производят конечный результат обработки объектов

Особенности Stream

1. 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

2. Стрим после обработки нельзя переиспользовать.

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

Операции со Stream

| Method | Description |
|--|---|
| <code><R, A> R collect(Collector<? super T, A, R> collectorFunc)</code> | Collects elements into a container, which is changeable, and returns the container. This is called a mutable reduction operation. Here, R specifies the type of the resulting container and T specifies the element type of the invoking stream. A specifies the internal accumulated type. The <i>collectorFunc</i> specifies how the collection process works. (Terminal operation.) |
| <code>long count()</code> | Counts the number of elements in the stream and returns the result. (Terminal operation.) |
| <code>Stream<T> filter(Predicate<? super T> pred)</code> | Produces a stream that contains those elements from the invoking stream that satisfy the predicate specified by <i>pred</i> . (Intermediate operation.) |
| <code>void forEach(Consumer<? super T> action)</code> | For each element in the invoking stream, the code specified by <i>action</i> is executed. (Terminal operation.) |
| <code><R> Stream<R> map(Function<? super T, ? extends R> mapFunc)</code> | Applies <i>mapFunc</i> to the elements from the invoking stream, yielding a new stream that contains those elements. (Intermediate operation.) |
| <code>DoubleStream mapToDouble(ToDoubleFunction<? super T> mapFunc)</code> | Applies <i>mapFunc</i> to the elements from the invoking stream, yielding a new DoubleStream that contains those elements. (Intermediate operation.) |

| | |
|--|---|
| <code>IntStream mapToInt(ToIntFunction<? super T> mapFunc)</code> | Applies <i>mapFunc</i> to the elements from the invoking stream, yielding a new IntStream that contains those elements. (Intermediate operation.) |
| <code>LongStream mapToLong(ToLongFunction<? super T> mapFunc)</code> | Applies <i>mapFunc</i> to the elements from the invoking stream, yielding a new LongStream that contains those elements. (Intermediate operation.) |
| <code>Optional<T> max(Comparator<? super T> comp)</code> | Using the ordering specified by <i>comp</i> , finds and returns the maximum element in the invoking stream. (Terminal operation.) |
| <code>Optional<T> min(Comparator<? super T> comp)</code> | Using the ordering specified by <i>comp</i> , finds and returns the minimum element in the invoking stream. (Terminal operation.) |
| <code>T reduce(T identityVal, BinaryOperator<T> accumulator)</code> | Returns a result based on the elements in the invoking stream. This is called a reduction operation. (Terminal operation.) |
| <code>Stream<T> sorted()</code> | Produces a new stream that contains the elements of the invoking stream sorted in natural order. (Intermediate operation.) |
| <code>Object[] toArray()</code> | Creates an array from the elements in the invoking stream. (Terminal operation.) |

Пример работы со Stream

При работа со Stream довольно часто используются lambda выражения и ссылки на методы:

```
public static void main(String[] args) {  
    long count = Stream  
        .generate(() -> new Random().nextInt(255))  
        .limit(100)  
        .filter(intValue -> intValue != 0)  
        .map(intValue -> (char) intValue.intValue())  
        .peek(System.out::print)  
        .count();    //завершающая операция  
    System.out.println(count);  
}
```

Вопросы

