Понятие потока данных в Stream API определеяется как канал передачи данных, позволяет использовать функциональный стиль при работе с разными структурами данных. Поток данных представляет собой последовательность объектов.

Поток не хранит сами данные, а только перемещает и обрабатывает неким образом. Данные поток получает из источника, чаще всего это коллекции, но не обязательно.

**Преимущества**

1. Зачастую, обработка выполняется параллельно, что повышает производительность.
2. Программисту не нужно заботится о подробностях реализации, параллельная обработка зашита в сами стримы, что повышает надежность, и исключает ошибки многопоточности.
3. Отложенное выполнение. Стримы начинают работу в конце, после вызова терминального метода. Никогда не вычисляются элементы данных, которые не нужны для выполнения завершающей операции.

Например stream().filter().limit(5) не будет фильтровать все элементы, а только те, что попадут в лимит 5.

Такие отложенные операции делают возможной работу с бесконечными потоками.

!Stream одноразовый, мы не можем его переиспользовать для различного набора операций (разветвить). Стрим не хранит данные, после того как он отработал, становится непригодным (IllegalStateException).

В основе иерархии strem API лежит интерфейс BaseStream

*Interface BaseStream<T, S extends BaseStream<T, S>>*

Здесь параметр T обозначает тип элементов, а S – тип патока данных, расширяющего интерфейс BaseStream.

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

* void close(): закрывает поток
* boolean isParallel(): возвращает true, если поток является параллельным
* Iterator<Т> iterator(): возвращает ссылку на итератор потока
* Spliterator<Т> spliterator(): возвращает ссылку на сплитератор 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. Как правило, работает быстрее, особенно если поток паралельный.

От BaseStream наследуется интерфейс *Stream<T>*, где Т – тип элемента в потоке данных.

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

Получить поток можно несколькими способами:

1. Метод *stream(),* объявленный в интерфейсе Collection

*Default Stream<E> stream()*;

В реализации по умолчанию возвращает последовательный поток данных.

1. Метод *parallelStream()*:

*Default Stream<E> parallelStream()*

Возвращает параллельный поток данных. Если получить параллельный поток нельзя, то вместо него может быть возвращен последовательный.

1. Последовательный Поток данных можно получать из массива, используя статический метод класса *Arrays*

*Static <T> Stream<T> stream(T[] arr)*

1. Пустой стрим

*Stream.empty()*;

1. Стрим из указанных элементов

*Stream.of(“1”, “2”, “3”)*

1. Стрим из файла, каждая новая строка становится элементом

*Stream<String> fromFileS = Files.lines(Paths.get("doc.txt"));*

1. Имеется возможность создавать бесконечные стримы

* *Stream.iterate(*1, m -> m + 1*)* мы задаем начальное значение, а также указываем, как будем получать следующий элемент. Использует UnaryOperator
* *Stream.generate(() -> “f5”)* позволяет бесконечно генерировать постоянные и случайные значения, которые соответствуют указанному выражению. Использует интерфейс supplier.

Операции со стримами делятся на **промежуточные** (конвеерные) и **терминальные**.

*Промежуточные* операции возвращают трансформированный поток. Над этим потоком затем можно выполнять другие операции.

|  |  |
| --- | --- |
| filter(Predicate<?> super T predicate) | Отработает как фильтр, вернет значения, которые удовлетворяют условию  myStream.filter((a) -> a.length() == 5); |
| sorted(Comparator<? super T> comparator) | Сортирует элементы в естественном порядке. Можно использовать компаратор  myStream.sorted(); myStream.sorted((a, b) -> a.length() - b.length()); myStream.sorted(String::compareTo); |
| limit(long maxSize) | Ограничить количество элементов указанным числом. Взять первых 3 элемента  myStream.limit(3); |
| skip(long n) | Пропустить указанное количество элементов. Взять все, кроме первых 3  myStream.skip(3); |
| distinct | Удаляет дубликаты элементов  myStream.distinct(). |
| peek(Consumer<? super T> action) | Выполняет действие над элементами, и возвращает сами элементы. Используется чтобы поменять какое-либо состояние объекта.  users.stream().peek(user -> user.setAge(user.getAge() + 1)); |
| map(Function<? Super T, ? extends R> функция отображения) | Выполняет действия над элементом, и возвращает результат. Функциональный интерфейс Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R.  users.stream().map(User::getAge) |
| flatMap(Function<? super T, ? extends Stream<? extends R>> mapper) | Преобразование одного элемента в 0 или несколько других. Например получить хобби пользователей (у каждого может быть несколько а может и не быть)  users.stream().map(User::getHobbies).flatMap(hobbies -> hobbies.stream()) |
| mapToInt(ToIntFunction<? super T> mapper), mapToDouble, mapToLong | Применяют функцию отображения, и создают новый числовой поток  users.stream().mapToDouble(user -> (double) user.getAge() / 100)  users.stream().mapToInt(user -> user.getHobbies().size()) |

Терминальные операции возвращают конечный результат в виде единственного элемента.

|  |  |
| --- | --- |
| findFirst | Вернет первый элемент стрима  users.stream().findFirst() |
| findAny | Вернет любой элемент стрима  users.stream().findAny() |
| collect | Собрать результаты обработки в коллекцию, либо еще куда-то  users.stream().filter((User user) -> user.getAge() > 50).collect(Collectors.*toList*()); |
| long count | Посчитать количество элементов в стриме  users.stream().filter((User user) -> user.getAge() > 50).count() |
| anyMatch(Predicate<? super T> predicate) | Возвращает True когда хотя бы один элемент удовлетворяет условию.  users.stream().anyMatch(user -> user.getName().equals("Vlad")); |
| noneMatch(Predicate<? super T> predicate) | Возвращает True когда ни один элемент не удовлетворяет условию  users.stream().noneMatch(user -> user.getAge() == 500); |
| allMatch(Predicate<? super T> predicate) | Возвращает True когда все элементы удовлетворяет условию  users.stream().allMatch(user -> user.getAge() > 0); |
| min(Comparator<? Super T>) | Найдет минимальный элемент, используя переданный компаратор  myStream.min(String::compareTo) |
| max(Comparator<? Super T>) | Найдет максимальный элемент, используя переданный компаратор  myStream.max(String::compareTo) |
| forEach(Consumer<? super T действие) | Применяет функцию ко всем элементам в рандомном порядке  users.stream().forEach(System.*out*::println); |
| forEachOrdered(Consumer<? super T> action) | Применяет функцию ко всем элементам по очереди  users.stream().forEachOrdered(System.*out*::println); |
| toArray(IntFunction<A[]> generator) | Приведет значения стрима к массиву  users.stream().toArray(User[]::new); |
| reduce(T identityVal, BinaryOperator<T> накопитель) | Преобразует все эллементы стрима в один объект. В identity накапливается результат, применяя функцию accumulator.  users.stream().map(User::getAge).reduce((a, b) -> a + b); |

Конвеерных операций в стриме может быть много, терминальный только один. После выполнения терминального метода стрим завершается, и мы не можем его больше использовать. При попытке использовать завершенный стрим возникает исключение IllegalStateException.

Конвеерные операции ленивые, обработка в них не начнется до тех пор, пока не будет вызван терминальный метод.

**reduce**

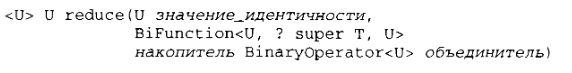
reduce – операция сведения элементов стрима в один элемент, имеет 3 варианта:

1. 

В качестве первого значения берется первый элемент потока. Накопитель представляет из себя *BinaryOperator*, у него есть метод T apply(T val1, T val2). Последовательно применяет какую-то операцию к текущему результату, и следующему элементу стрима.

1. 

Во втором варианте можно указать начальное значение identity. Возвращаемый тип T, так как в любом случае вернется хотя бы значение identity, если стрим пустой.

1. 

Последняя форма используется в параллельных стримах. У нее есть параметр объединитель. То есть. Сначала элементы разбиваются на группы, с ними параллельно выполняется какая-то операция. Затем они объединяются, с помощью операции объединителя, которая может быть другой.

// получить сумму квадратов элементов с помощью параллельного стрима и reduce  
Stream<Integer> numStream = numbers.parallelStream();  
// неверное решение, без объединителя  
int res = numStream.reduce(0, (a, b) -> a + b\*b);  
System.*out*.println(res);  
// верное, при объединении нам нужно просто слаживать числа  
res = numbers.parallelStream().reduce(0, (a, b) -> a + b\*b, (a, b) -> a + b);  
System.*out*.println(res);

**COLLECTORS**

Чтобы преобразовать поток в коллекцию есть метод **collect(Collector<? super T,A,R> collector)**.

Параметр R представляет тип результата метода, параметр Т - тип элемента в потоке, а параметр А - тип промежуточных накапливаемых данных. В итоге параметр collector представляет функцию преобразования потока в коллекцию.

Java предоставляет ряд встроенных функций, определенных в классе Collectors.

* toList – стрим приводится к списку;
* toCollection – получаем коллекцию;

Метод **toCollection()**принимает [лямбда-выражение](https://vertex-academy.com/tutorials/ru/?p=5473&preview=true) типа **поставщик (Supplier)**, которое должно вернуть коллекцию, в которую мы хотим сохранить данные.

* toSet – получаем множество;
* toConcurrentMap, toMap – преобразует в Map;
* summingInt, summingDouble, summingLong – если требуется получить сумму чисел.

Integer a = users.stream().collect(Collectors.*summingInt*(User::getAge));

* averagingInt, averagingDouble, averagingLong – вернуть среднее значение чисел;

Double a = users.stream().collect(Collectors.*averagingInt*(User::getAge));

* groupingBy – если необходимо разбить коллекцию на части, сгруппировать по какому-либо признаку. возвращает Map<N, List<T>>;

Map<Integer, List<String>> list = users.stream().flatMap((user) -> user.getHobbies().stream()).collect(Collectors.*groupingBy*(String::length));

* partitioningBy – делит элементы на группы true либо false в соответствии с условием.

Map<Boolean, List<String>> list = users.stream().flatMap((user) -> user.getHobbies().stream()).collect(Collectors.*partitioningBy*((s) -> s.length()>5));

* Collectors.counting() применяется в groupingBy для подсчета элементов в группе

Map<Integer, Long> list = users.stream().flatMap((user) -> user.getHobbies().stream()).collect(Collectors.*groupingBy*(String::length, Collectors.*counting*()));

* mapping позволяет преобразовать элементы в другой тип

List<Integer> a = users.stream().collect(Collectors.*mapping*(User::getAge, Collectors.*toList*()));

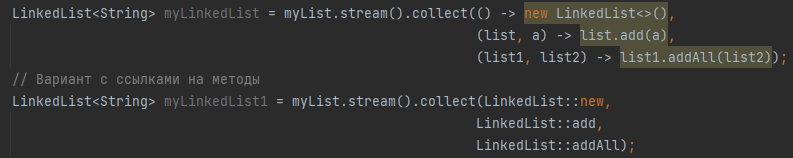
Есть другая форма метода collect, которая представляет больше возможностей:



*Адресат –* поставщик каких-то объектов, можно указать ссылку на конструктор;

*Накопитель –* добавляет элемент к результату;

*Объединитель –* объединяет два частичных результата. (Накопитель и объединитель действуют аналогично reduce).



Реализация своего коллектора



Интерфейс Collector требует реализации пяти методов. **Supplier** возвращает лямбда-выражение, создающее контейнер для хранения промежуточных выражений.

**Accumulator** добавляет очередное значение в контейнер промежуточных значений. Если быть точным, то accumulator возвращает лямбда-выражение, которое обрабатывает очередное значение и сохраняет его.

**Combiner** возвращает лямбда-выражение, объединяющее два контейнера промежуточных значений в один. Дело в том, что Stream API может создать несколько таки контейнеров, для параллельной обработки и в конце слить их в один общий контейнер.

**Finisher** возвращает лямбда-выражение, которое производит финальное преобразование: обрабатывает содержимого контейнера промежуточных результатов и приводит его к заданному выходному типу.

**Когда следует применять параллельные потоки**

Промежуточные операции могут выполняться с сохранением и без сохранения состояния. Например метод sorted() выполняется с сохранением состояния, а метод filter без.

**Ассоциативность** – операция ассоциативна когда не важен порядок действий.

**Операция без вмешательства** – означает, что источник данных не видоизменяется самой операцией.

Как правило, любая операция в параллельном потоке должна выполняться без сохранения состояния, быть ассоциативной и без вмешательств. Этим гарантируется, что результаты выполнения операций в параллельном потоке данных остаются такими же, как и в последовательном.

При выполнении метода *forEach()*  упорядоченность параллельного потока может не сохранятся. Если требуется сохранить упорядоченность, то лучше воспользоваться методом *forEachOrdered()*.

**OPTIONAL**

Optional может содержать значение, а может быть пустым.

* Через метод isPresent можно проверить наличие в нем значения.
* Получить значение можем через метод get(). Если объекта нет, выбрасывает NoSuchElementException.
* Метод orElse() возвращает обернутое значение, если оно есть. В противном случае возвращается переданное значение.
* orElseGet() аналогичен orElse, но принимает параметром функциональный интерфейс поставщика.
* orElseThrow выбрасывает исключение, если значение отсутствует.
* map(), flatMap() используются для преобразования объекта в другой объект. Если переданная функция возвращает null, to map вернет пустой Optional. Разница между ними в том, что в map передается метод, возвращающий объект, а в flatMap - Optional

**Итераторы и потоки данных**

Для перебора элементов в потоке поддерживаются 2 типа итераторов.

**Обычный итератор** применяется почти так же, как и в коллекции. Имеет методы:

* **hasNext()** – возвращает true если есть следующий элемент, иначе false;
* **next()** – возвращает следующий элемент.

Есть дополнительные типы итераторов для обработки потоков примитивных типов данных: **PrimiriveIterator, PrimiriveIterator.ofDouble, PrimiriveIterator.ofLong, PrimiriveIterator.ofInt.** Они действуют так же, как обычный итератор.

Чтобы получить итератор нужно вызвать метод *itearator()*.

**Spliterator** служит альтернативой обычному итератору, особенно для параллельной обработки. Имеет методы:

* **tryAdvance(Consumer<? Super T> действие)** – возвращает true, если имеется следующий элемент, иначе false. Заданное действие будет выполняться над следующим элементом потока.
* **forEachRemaining(Consumer<? Super T> действие)** – Выполняет действие над всеми элементами сразу. Этот метод исключает потребность в циклическом переборе элементов.
* **trySplit()** – разделяет итерируемые элементы на 2 части, возвращая новый Spliterator для одной части, а другая часть остается доступной исходному итератору. Если разделить Spliterator не удается, возвращается null.