**Hướng dẫn và giải thích nguyên lý làm việc của Stream trong Java 8**

Stream là một trong những concept được coi là đem sự thay đổi lớn nhất trong Java 8.

[Functional Interfaces & Lambda Expressions](https://loda.me/java-8-functional-interfaces-and-lambda-expressions-cuc-de-hieu-loda1553649037877)

[Optional](https://loda.me/java-8-optional-loda1553606615611)

**Khái quát**

Stream là một abtract layer cho phép bạn xử lý một dòng dữ liệu dựa trên các thao tác đã định nghĩa trước.

Bạn có thể tạo Stream từ các nguồn dữ liệu như Collections, Arrays hoặc I/O resources.

Collection<String> collection = Arrays.asList("hello", "loda", "kaka");

Stream<String> streamOfCollection = collection.stream(); // Tạo ra một stream từ collection

Mặc định các lớp kế thừa của Collection đều có hàm .stream():

List<String> list = new ArrayList<>();

Stream<String> stream = list.stream(); // tạo ra 1 luồng

Stream<String> parallelStream = list.parallelStream(); // luồng dữ liệu song song (xử lý trên nhiều thread cùng lúc)

**Cách sử dụng**

Chức năng của Stream là cực kì đa dạng giúp bạn thao tác dữ liệu dễ dàng hơn.

**forEach**

Duyệt qua toàn bộ dữ liệu của bạn

list.stream().forEach(s -> System.out.println(s));

**map**

Tạo ra các giá trị mới từ dữ liệu hiện có

Arrays.asList(3, 5, 7)

.stream() // tạo ra Stream từ List<Integer>

.map(i -> "loda-"+i) // biến đổi từng phần tử thành String

.map(String::toUpperCase) // biến đổi từng phần tử thành Upper case

.forEach(System.out::println); // in ra xem thử

**filter**

filter() gíup chúng ta thao tác với những dữ liệu mong muốn

Arrays.asList(2, 3, 5, 7)

.stream()

.filter(i -> i % 2 != 0) //từ đây trở đi, chúng ta chỉ muốn làm việc với số lẻ

.map(i -> "loda-" + i)

.map(String::toUpperCase)

.forEach(System.out::println);

**limit**

Giới hạn số lượng dữ liệu cần xử lý

IntStream.range(1, 1000).boxed() // Tạo ra Stream có dữ liệu từ 1->999

.filter(i -> i % 2 != 0)

.map(i -> "loda-" + i)

.map(String::toUpperCase)

.limit(10) // Chúng ta giới hạn lấy 10 cái rồi in ra

.forEach(System.out::println);

**sorted**

sắp xếp Stream

IntStream.range(1, 1000).boxed() // Tạo ra Stream có dữ liệu từ 1->999

.filter(i -> i % 2 != 0)

.map(i -> "loda-" + i)

.map(String::toUpperCase)

.limit(10)

.sorted() // Sắp xếp dữ liệu đã xử lý

.forEach(System.out::println);

// OUTPUT:

/\*

LODA-1

LODA-11

LODA-13

LODA-15

\*/

// đây là vì dữ liệu là String, nó đang sort StringString

Bạn có thể tự định nghĩa cách sort bằng cách thêm Comparator vào

sorted((o1, o2) -> o1.compareTo(o2))

**collect**

collect giúp chúng ta lấy toàn bộ dữ liệu đã biến đổi trong Stream thành đối tượng mình mong muốn

List<String> result = IntStream.range(1, 1000).boxed()

.filter(i -> i % 2 != 0)

.map(i -> "loda-" + i)

.map(String::toUpperCase)

.limit(10)

.sorted(Comparator.naturalOrder()) // một cách khác để sort

.collect(Collectors.toList());

**Xử lý song song**

List<String> result = IntStream.range(1, 1000).boxed()

.parallel() // tạo một Stream xử lý dữ liệu song song, tương đương với parallelStream()

.filter(i -> i % 2 != 0)

.map(i -> "loda-" + i)

.map(String::toUpperCase)

.limit(10)

.sorted(Comparator.naturalOrder()) // một cách khác để sort

.collect(Collectors.toList());

**Bản chất của Stream**

Bạn hãy chạy chương trình này nhé

List<String> result = Stream.of("bạn", "hãy", "like", "Fanpage", "loda","dể","cập","nhật","nhiều","hơn")

.filter(s -> {

System.out.println("[filtering] " + s);

return s.length()>=4;

})

.map(s -> {

System.out.println("[mapping] " + s);

return s.toUpperCase();

})

.limit(3)

.collect(Collectors.toList());

System.out.println("----------------------");

System.out.println("Result:");

result.forEach(System.out::println);

Kết quả:

[filtering] bạn // không thoả mãn

[filtering] hãy // tiếp tục tìm, cũng k thoả mãn

[filtering] like // thoả mãn

[mapping] like // mapping nó luôn

[filtering] Fanpage // lại quay lại filter tìm tiếp, thoả mãn

[mapping] Fanpage // mapping

[filtering] loda // thoả mãn

[mapping] loda // mapping

// Đủ 3 trường hợp thoả mãn, dừng.

----------------------

Result:

LIKE

FANPAGE

LODA

Bạn sẽ thấy rằng chương trình chỉ xử lý dữ liệu vừa đủ thoả mãn điều kiện limit(3) mà thôi, còn lại nó sẽ bỏ qua để tối ưu hoá performance.

Chứng tỏ Stream là Lazy evaluation. Hiểu đơn giản là nó sẽ không xử lý dữ liệu trực tiếp qua từng bước, mà chờ bạn khai báo xong tất cả các thao tác operation như map, filter,v.v.. cho tới khi gặp lệnh .collect() thì nó thực hiện toàn bộ trong một vòng lặp duy nhất.

Hàm .collect() và một số hàm như min(), max(), count() được gọi là terminal operation. Khi gọi những function có dạng terminal thì Stream mới chính thức hoạt động.

Một lưu ý khi sử dụng là **Stream không được tái sử dụng**. Ví dụ:

Stream<String> stream =

Stream.of("loda", ".", "me","like").filter(element -> element.contains("e"));

Optional<String> anyElement = stream.findAny(); //Lấy ra một phần tử bất kỳ trong Stream, nó sẽ trả ra Optional

// Thực hiện dòng lệnh tiếp theo sẽ bắn ra IllegalStateException

Optional<String> firstElement = stream.findFirst();

Vì Stream được tạo ra để **xử lý** dữ liệu chứ không phải để **lưu trữ**!

Nên muốn sử dụng, mỗi lần bạn sẽ cần tạo ra 1 Stream mới.