Theory: Infinite streams

12 minutes 0 / 4 problems solved

Collections in Java always have a finite number of elements because the memory of a computer is limited. However, Stream API allows us to operate on infinite streams of elements. It is possible due to the lazy nature of streams: creating an infinite stream is an intermediate operation, so it actually does not create elements until a terminal operation is executed. It is supposed that a stream has already become finite at this moment.

In this topic, we will learn how to create infinite streams by using `generate` and `iterate` methods, and how they can be used.

§1. Generating a stream of objects

The first method we consider is `generate`. It takes a **supplier function** that produces an object that'll be a part of the stream.

```
Stream<T> generate(Supplier<T> supplier)
```

In the example below, we create an infinite stream of random numbers by using the reference to the `random` method of the `Math` class.

```
Stream<Double> randomNumbers = Stream.generate(Math::random);
```

The supplier function can be a constructor of a class as well:

```
Stream<User> userStream = Stream.generate(User::new);
```

It generates a stream of `User` objects employing its no-argument constructor.

Depending on how the **supplier function** works all the elements of such a stream may be different or the same.

But why don't we try to print the elements of an infinite stream by using the `forEach` method? Well, it's not a good idea. Your computer simply does not have enough memory to do that.

> No matter how you've created an infinite stream you have to make it finite by using `limit` before invoking a terminal operation or to invoke only special kind of terminal operations like `findFirst` / `findAny` that does not try to generate all the elements in the infinite stream.

In the following example, we create an infinite stream and take only the first five elements from it:

```
Stream.generate(Math::random)
        .limit(5) // don't miss it
        .forEach(System.out::println);
```

It generates and prints five random numbers. Your output may look like the following:

```
0.9700910364529439
0.04347176223398197
0.2887419669771004
0.32376617731162183
0.22532773260604255
```

Of course, we can also use a variable to produce an arbitrary number of elements, instead of a specific constant.

Let's look at another example that applies the `findAny` method. This method just returns a single element from a stream.

Current topic:

Infinite streams

Topic depends on:

× Functional data processing with streams

Table of contents:

```
double randomNumber = Stream.generate(Math::random).findAny().get();
```

This code works perfectly because the `findAny` method causes only one element to be generated. It doesn't try to create the whole infinite stream.

§2. Iterations

The second method we consider is `iterate`. It is a complete alternative to standard loops.

It takes a starting value (a **seed**) and an **operator function** to generate the next value based on the previous one.

```
Stream<T> iterate(T seed, UnaryOperator<T> next)
```

As an example, here is an infinite stream of odd numbers:

```
Stream<Integer> oddNumbersStream = Stream.iterate(1, x -
> x + 2); // 1, 3, 5, ...
```

Here the starting value is `1`, and the function to produce the next values is `x -> x + 2`.

Similarly to the `generate` method, this stream is infinite. If we'd like to obtain the values, we should make this stream finite.

In the following example, we print the first five odd numbers using the `iterate` method.

```
Stream.iterate(1, x -> x + 2)
        .limit(5)
        .forEach(System.out::println); // 1 3 5 7 9
```

If we use the `findFirst` method as the terminal operation instead of `forEach`, we can skip the `limit` operation because it is obvious that the first element is 1 (the seed).

Interestingly, the following code will lead to an error. Try to guess why!

```
int min = Stream.iterate(1, x -
> x + 1).min(Comparator.naturalOrder()).get();
```

To us, it is obvious that the smallest element here is 1. But the computer does not know the properties of the function that is used to produce the next values and cannot comprehend that the function will not create another minimum value. The same mistake will occur if we try to perform any operation that formally requires to process of all elements of the infinite stream.

In addition, the `iterate` method has an overloaded version that generates a finite stream. It takes a **predicate** to determine whether the stream has the next element.

```
Stream<T> iterate(T seed, Predicate<T> hasNext, UnaryOperator<T> next)
```

In the following example, we use this method to print the odd numbers that are less than 10:

```
Stream.iterate(1, x -> x < 10, x -> x + 2)
        .forEach(System.out::println); // 1 3 5 7 9
```

Look! It is a complete alternative to the classic `for` loop.

```
for (int i = 1; i < 10; i += 2) {
    System.out.println(i);
}
```

By this, we are completing our consideration of infinite streams and methods to produce them.

§3. Conclusion

We have considered two methods to produce infinite streams: `generate` and `iterate`. The first one generates a stream of objects, while the second one creates iterations by using a counter and can be used as an alternative to loops. When working with infinite streams, you must not forget to limit them or should only use suitable terminal operations that do not use all elements of the stream.

Report a typo

33 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion