모듈 java.base
패키지 java.util.stream

인터페이스 Stream <T>

- 유형 매개 변수 :
  T -스트림 요소의 유형

  모든 수퍼 인터페이스 :
  AutoCloseable, BaseStream<T,Stream<T>>

공중 인터페이스 스트림 <T가>
연장 BaseStream <T, 트림 <T >>

순차 및 병렬 집계 작업을 지원하는 일련의 요소입니다. 다음의 예는 사용하는 전체 동작을 도시 Stream하고 IntStream:

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

이 예에서, widgetsA는 Collection<Widget>. Widget를 통해 객체 스트림을 만들고 Collection.stream()필터링하여 빨간색 위젯 만 포함하는 스트림을 생성 한 다음 int각 빨간색 위젯의 가중치를 나타내는 값 스트림으로 변환합니다 . 그런 다음이 스트림을 합하여 총 가중치를 생성합니다.

이외에 Stream오브젝트 레퍼런스들의 스트림이며, 프리미티브 전문화 존재 IntStream, LongStream및 DoubleStream"스트림"이라하고 준수 특성 및 제한은 여기에 설명되어 모두는.

계산을 수행하기 위해 스트림 작업 이 스트림 파이프 라인 으로 구성됩니다 . 스트림 파이프 라인은 소스 (배열, 컬렉션, 생성기 함수, I / O 채널 등일 수 있음), 0 개 이상의 중간 작업 (예 : 스트림을 다른 스트림으로 변환 filter(Predicate)) 및 조작 단말 (결과 또는 부작용, 예컨대 생산 count()이상 forEach(Consumer)). 스트림은 게으르다. 소스 데이터에 대한 계산은 터미널 작업이 시작될 때만 수행되고 소스 요소는 필요한 경우에만 사용됩니다.

스트림 구현은 결과 계산을 최적화하는 데 상당한 위도를 허용합니다. 예를 들어, 스트림 구현은 스트림 파이프 라인에서 작업 (또는 전체 단계)을 자유롭게 제거 할 수 있으며, 따라서 계산 결과에 영향을 미치지 않는다는 것을 증명할 수있는 경우 동작 매개 변수의 호출을 제거 할 수 있습니다. 즉, 동작 매개 변수의 부작용이 항상 실행되는 것은 아니며 달리 지정되지 않는 한 의존해서는 안됩니다 (예 : 터미널 작업 forEach및 forEachOrdered). (이러한 최적화의 구체적인 예는 count()작업 에 대한 API 노트를 참조하십시오 . 자세한 내용은 스트림 패키지 문서의 부작용 섹션을 참조하십시오 .)

컬렉션과 스트림은 표면적 유사성을 가지고 있지만 목표는 다릅니다. 컬렉션은 주로 해당 요소의 효율적인 관리 및 액세스와 관련됩니다. 반대로 스트림은 요소에 직접 액세스하거나 요소를 조작 할 수있는 수단을 제공하지 않으며 대신 소스와 해당 소스에서 집계로 수행 될 계산 작업을 선언적으로 설명하는 것과 관련이 있습니다. 그러나 제공된 스트림 작업이 원하는 기능을 제공하지 않는 경우 BaseStream.iterator()및 BaseStream.spliterator()작업을 사용하여 제어 된 순회를 수행 할 수 있습니다.

위의 "위젯"예제와 같은 스트림 파이프 라인 은 스트림 소스에 대한 쿼리 로 볼 수 있습니다 . 소스가 동시 수정 (예 :)을 위해 명시 적으로 설계된 경우가 ConcurrentHashMap아니면 쿼리하는 동안 스트림 소스를 수정하면 예측할 수 없거나 잘못된 동작이 발생할 수 있습니다.

대부분의 스트림 작업 은 위의 예에서 w -> w.getWeight()전달 된 람다 식과 같이 사용자 지정 동작을 설명하는 매개 변수를 허용합니다 mapToInt. 올바른 동작을 유지하기 위해 다음 동작 매개 변수는 다음과 같습니다.

- 간섭이 없어야합니다 (스트림 소스를 수정하지 않음). 과
- 대부분의 경우 상태 비 저장 이어야합니다 (그 결과는 스트림 파이프 라인 실행 중에 변경 될 수있는 상태에 의존해서는 안됩니다).

이러한 매개 변수는 항상 과 같은 기능적 인터페이스의 인스턴스이며, Function종종 람다 식 또는 메서드 참조입니다. 달리 지정하지 않는 한 이러한 매개 변수는 null 이 아니 어야합니다 .

스트림은 한 번만 작동해야합니다 (중간 또는 터미널 스트림 작업 호출). 예를 들어 동일한 소스가 두 개 이상의 파이프 라인을 공급하는 "포크 된"스트림 또는 동일한 스트림의 여러 순회를 제외합니다. IllegalStateException 스트림이 재사용되고 있음을 감지하면 스트림 구현이 throw 될 수 있습니다 . 그러나 일부 스트림 작업은 새 스트림 객체가 아닌 수신자를 반환 할 수 있으므로 모든 경우에 재사용을 감지하지 못할 수 있습니다.

스트림에는 BaseStream.close()메서드가 있고 AutoCloseable. 닫힌 후 스트림에서 작동하면 IllegalStateException. 대부분의 스트림 인스턴스는 특별한 리소스 관리가 필요하지 않은 컬렉션, 배열 또는 생성 함수에 의해 지원되므로 사용 후 실제로 닫을 필요가 없습니다. 일반적으로에서 반환 된 것과 같이 소스가 IO 채널 인 스트림 만 Files.lines(Path)닫아야합니다. 스트림을 닫아야하는 경우 try-with-resources 문 또는 유사한 제어 구조 내에서 리소스로 열어야 작업이 완료된 후 즉시 닫힙니다.

스트림 파이프 라인은 순차적으로 또는 병렬로 실행될 수 있습니다 . 이 실행 모드는 스트림의 속성입니다. 스트림은 순차 또는 병렬 실행의 초기 선택으로 생성됩니다. (예를 들어, Collection.stream()순차 스트림을 Collection.parallelStream()생성하고 병렬 스트림을 생성합니다.)이 실행 모드 선택은 BaseStream.sequential() 또는 BaseStream.parallel()메소드에 의해 수정 될 수 있으며 메소드로 쿼리 될 수 있습니다 BaseStream.isParallel().

이후:
1.8

또한보십시오:
IntStream, LongStream, DoubleStream, java.util.stream

- ### 중첩 된 클래스 요약

중첩 클래스

수정 자 및 유형	상호 작용	기술
static interface	Stream.Builder<T>	위한 가변 빌더 Stream.

방법 요약

모든 방법 정적 방법 인스턴스 방법 추상 방법 기본 방법

수정 자 및 유형	방법	기술
boolean	allMatch(Predicate<? super T> predicate)	이 스트림의 모든 요소가 제공된 술어와 일치하는지 여부를 리턴합니다.
boolean	anyMatch(Predicate<? super T> predicate)	이 스트림의 요소가 제공된 술어와 일치하는지 여부를 리턴합니다.
static <T> Stream.Builder<T>	builder()	에 대한 빌더를 반환합니다 Stream.
<R> R	collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)	이 스트림의 요소에 대해 가변 감소 작업을 수행 합니다.
<R,A> R	collect(Collector<? super T,A,R> collector)	를 사용하여이 스트림의 요소에 대해 가변 감소 작업을 수행합니다 Collector.
static <T> Stream<T>	concat(Stream<? extends T> a, Stream<? extends T> b)	첫 번째 스트림의 모든 요소와 두 번째 스트림의 모든 요소가 뒤 따르는 요소가있는 느리게 연결된 스트림을 만듭니다.
long	count()	이 스트림의 요소 수를 리턴합니다.
Stream<T>	distinct()	이 스트림의 고유 요소 (에 따라 Object.equals(Object)) 로 구성된 스트림을 리턴 합니다.
default Stream<T>	dropWhile(Predicate<? super	이 스트림이 정렬 된 경우 지정된 조건 자와

	T> predicate)	일치하는 요소의 가장 긴 접두사를 삭제 한 후 이 스트림의 나머지 요소로 구성된 스트림을 반환합니다.
static <T> Stream<T>	empty()	빈 순차를 반환합니다 Stream.
Stream<T>	filter(Predicate<? super T> predicate)	주어진 술어와 일치하는이 스트림의 요소로 구성된 스트림을 리턴합니다.
Optional<T>	findAny()	Optional스트림의 일부 요소를 설명하거나 Optional스트림이 비어 있는 경우 비어 있는 것을 리턴합니다 .
Optional<T>	findFirst()	Optional이 스트림의 첫 번째 요소를 설명하거나 Optional스트림이 비어 있는 경우 비어 있는 것을 리턴합니다 .
<R> Stream<R>	flatMap(Function<? super T,? extends Stream<? extends R>> mapper)	이 스트림의 각 요소를 제공된 매핑 함수를 각 요소에 적용하여 생성 된 매핑 된 스트림의 내용으로 대체 한 결과로 구성된 스트림을 반환합니다.
DoubleStream	flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)	DoubleStream이 스트림의 각 요소를 제공된 매핑 함수를 각 요소에 적용하여 생성 된 매핑 된 스트림의 내용으로 대체 한 결과로 구성된를 반환합니다 .
IntStream	flatMapToInt(Function<? super T,? extends IntStream> mapper)	IntStream이 스트림의 각 요소를 제공된 매핑 함수를 각 요소에 적용하여 생성 된 매핑 된 스트림의 내용으로 대체 한 결과로 구성된를 반환합니다 .
LongStream	flatMapToLong(Function<? super T,? extends LongStream> mapper)	LongStream이 스트림의 각 요소를 제공된 매핑 함수를 각 요소에 적용하여 생성 된 매핑 된 스트림의 내용으로 대체 한 결과로 구성된를 반환합니다 .
void	forEach(Consumer<? super T> action)	이 스트림의 각 요소에 대해 작업을 수행합니다.
void	forEachOrdered(Consumer<? super T> action)	스트림에 정의 된 발생 순서가있는 경우 스트림의 발생 순서대로이 스트림의 각 요소에 대한 작업을 수행합니다.
static <T> Stream<T>	generate(Supplier<? extends T> s)	제공된에 의해 각 요소가 생성되는 무한 순차 비 순차 스트림을 반환합니다 Supplier.
static <T> Stream<T>	iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)	주어진 술어 를 만족시키는 조건으로 Stream주어진 next함수를 초기 요소 에 반복적으로 적용하여 생성 된 순차 순서를 반환합니다 hasNext.
static <T> Stream<T>	iterate(T seed, UnaryOperator<T> f)	무한 시퀀스 주문시 돌려 Stream함수의 반복적 적용에 의해 생성 된 f초기 요소 seed생산 Stream이루어진 seed, f(seed), f(f(seed))등
Stream<T>	limit(long maxSize)	이 스트림의 요소로 구성된 스트림을 리턴하며 maxSize길이 가 더 길지 않도록 잘립니다 .
<R> Stream<R>	map(Function<? super T,? extends R> mapper)	지정된 함수를이 스트림의 요소에 적용한 결과로 구성된 스트림을 리턴합니다.
DoubleStream	mapToDouble(ToDoubleFunction<? super T> mapper)	DoubleStream지정된 함수를이 스트림의 요소에 적용한 결과로 구성된를 리턴 합니다.
IntStream	mapToInt(ToIntFunction<? super T> mapper)	IntStream이 스트림의 요소에 지정된 함수를 적용한 결과로 구성된를 리턴 합니다.
LongStream	mapToLong(ToLongFunction<? super T> mapper)	LongStream지정된 함수를이 스트림의 요소에 적용한 결과로 구성된를 리턴 합니다.
Optional<T>	max(Comparator<? super T> comparator)	제공된에 따라이 스트림의 최대 요소를 반환합니다 Comparator.

Optional<T>	min(Comparator<? super T> comparator)	제공된에 따라이 스트림의 최소 요소를 반환합니다 Comparator.
boolean	noneMatch(Predicate<? super T> predicate)	이 스트림의 요소가 제공된 술어와 일치하지 않는지 여부를 리턴합니다.
static <T> Stream<T>	of(T t)	Stream단일 요소를 포함 하는 순차 를 반환합니다 .
static <T> Stream<T>	of(T... values)	요소가 지정된 값인 순차적으로 정렬 된 스트림을 반환합니다.
static <T> Stream<T>	ofNullable(T t)	Streamnull이 아닌 경우 단일 요소를 포함 하는 순차 를 반환하고 , 그렇지 않으면 빈을 반환합니다 Stream.
Stream<T>	peek(Consumer<? super T> action)	이 스트림의 요소로 구성된 스트림을 반환하고 결과 스트림에서 요소가 소비 될 때 각 요소에 대해 제공된 작업을 추가로 수행합니다.
Optional<T>	reduce (BinaryOperator<T> accumulator)	연관 누적 함수를 사용하여이 스트림의 요소에 대해 감소 를 수행하고 감소 된 값 (있는 경우)을 설명하는을 리턴 합니다.Optional
T	reduce(T identity, BinaryOperator<T> accumulator)	Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<U> U	reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T>	skip(long n)	Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	sorted()	Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<T>	sorted(Comparator<? super T> comparator)	Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
default Stream<T>	takeWhile(Predicate<? super T> predicate)	이 스트림이 정렬 된 경우 지정된 술어와 일치하는이 스트림에서 가져온 요소의 가장 긴 접두사로 구성된 스트림을 리턴합니다.
Object[]	toArray()	이 스트림의 요소를 포함하는 배열을 리턴합니다.
<A> A[]	toArray (IntFunction<A[]> generator)	제공된 generator함수를 사용하여 반환 된 배열을 할당하고 분할 된 실행이나 크기 조정에 필요할 수있는 추가 배열을 사용하여이 스트림의 요소를 포함하는 배열을 반환합니다 .

- **인터페이스 java.util.stream. BaseStream**

  close, isParallel, iterator, onClose, parallel, sequential, spliterator, unordered

- ○ **방법 세부 정보**
  - **필터**

    Stream < T > filter (Predicate <? super T > predicate)

주어진 술어와 일치하는이 스트림의 요소로 구성된 스트림을 리턴합니다.

이것은 중간 작업 입니다.

매개 변수 :
predicate- 포함해야하는지 결정하기 위해 각 요소에 적용 할 비 간섭 , 상태 비 저장 술어
보고:
새로운 스트림

- **지도**

<R> Stream <R> map (Function <? super T ,? extends R> mapper)

지정된 함수를이 스트림의 요소에 적용한 결과로 구성된 스트림을 리턴합니다.

이것은 중간 작업 입니다.

유형 매개 변수 :
R -새 스트림의 요소 유형
매개 변수 :
mapper- 각 요소에 적용 할 비 간섭 , 상태 비 저장 함수
보고:
새로운 스트림

- **mapToInt**

IntStream mapToInt (ToIntFunction <? super T > 매퍼)

IntStream이 스트림의 요소에 지정된 함수를 적용한 결과로 구성된를 리턴 합니다.

이것은 중간 작업 입니다.

매개 변수 :
mapper- 각 요소에 적용 할 비 간섭 , 상태 비 저장 함수
보고:
새로운 스트림

- **mapToLong**

LongStream mapToLong (ToLongFunction <? super T > 매퍼)

LongStream지정된 함수를이 스트림의 요소에 적용한 결과로 구성된를 리턴 합니다.

이것은 중간 작업 입니다.

매개 변수 :
mapper- 각 요소에 적용 할 비 간섭 , 상태 비 저장 함수
보고:
새로운 스트림

- **mapToDouble**

DoubleStream mapToDouble (ToDoubleFunction <? super T > 매퍼)

DoubleStream지정된 함수를이 스트림의 요소에 적용한 결과로 구성된를 리턴 합니다.

이것은 중간 작업 입니다.

매개 변수 :
mapper- 각 요소에 적용 할 비 간섭 , 상태 비 저장 함수
보고:
새로운 스트림

- **flatMap**

```
<R>  Stream <R> flatMap ( Function <? super T ,? extends Stream <? extends R >> mapper)
```

이 스트림의 각 요소를 제공된 매핑 함수를 각 요소에 적용하여 생성 된 매핑 된 스트림의 내용으로 대체 한 결과로 구성된 스트림을 반환합니다. 매핑 된 각 스트림은 closed해당 내용이이 스트림에 배치 된 후입니다. (매핑 된 스트림이 null 빈 스트림 인 경우 대신 사용됩니다.)

이것은 중간 작업 입니다.

API 참고 :

이 flatMap()작업은 스트림의 요소에 일대 다 변환을 적용한 다음 결과 요소를 새 스트림으로 병합하는 효과가 있습니다.

예.

경우 orders구매 주문의 흐름이며, 각 구매 주문 라인 항목의 컬렉션을 포함, 그 다음은 모든 주문의 모든 광고 항목이 포함 된 스트림을 생성합니다 :

```
orders.flatMap(order -> order.getLineItems().stream())...
```

path파일의 경로 인 경우 다음은 words해당 파일에 포함 된 의 스트림을 생성 합니다.

```
Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8);
Stream<String> words = lines.flatMap(line -> Stream.of(line.split(" +")));
```

mapper전달 된 함수 flatMap는 간단한 정규식을 사용하여 한 줄을 단어 배열 로 분할 한 다음 해당 배열에서 단어 스트림을 만듭니다.

유형 매개 변수 :

R -새 스트림의 요소 유형

매개 변수 :

mapper- 새로운 값의 스트림을 생성하는 각 요소에 적용 할 비 간섭 , 상태 비 저장 함수

보고:

새로운 스트림

- **flatMapToInt**

```
IntStream  flatMapToInt ( Function <? super T ,? extends IntStream > mapper)
```

Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

This is an intermediate operation.

Parameters:

mapper - a non-interfering, stateless function to apply to each element which produces a stream of new values

Returns:

the new stream

See Also:

flatMap(Function)

- **flatMapToLong**

```
LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)
```

Returns an LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to

each element. Each mapped stream is `closed` after its contents have been placed into this stream. (If a mapped stream is `null` an empty stream is used, instead.)

This is an intermediate operation.

Parameters:
`mapper` - a non-interfering, stateless function to apply to each element which produces a stream of new values
Returns:
the new stream
See Also:
`flatMap(Function)`

- **flatMapToDouble**

```
DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)
```

Returns an `DoubleStream` consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is `closed` after its contents have placed been into this stream. (If a mapped stream is `null` an empty stream is used, instead.)

This is an intermediate operation.

Parameters:
`mapper` - a non-interfering, stateless function to apply to each element which produces a stream of new values
Returns:
the new stream
See Also:
`flatMap(Function)`

- **distinct**

```
Stream<T> distinct()
```

Returns a stream consisting of the distinct elements (according to `Object.equals(Object)`) of this stream.

For ordered streams, the selection of distinct elements is stable (for duplicated elements, the element appearing first in the encounter order is preserved.) For unordered streams, no stability guarantees are made.

This is a stateful intermediate operation.

API Note:
Preserving stability for `distinct()` in parallel pipelines is relatively expensive (requires that the operation act as a full barrier, with substantial buffering overhead), and stability is often not needed. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significantly more efficient execution for `distinct()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `distinct()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.
Returns:
the new stream

- **sorted**

```
Stream<T> sorted()
```

Returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not `Comparable`, a `java.lang.ClassCastException` may be thrown when the terminal operation is executed.

For ordered streams, the sort is stable. For unordered streams, no stability guarantees are made.

This is a stateful intermediate operation.

Returns:
the new stream

- **sorted**

`Stream<T> sorted(Comparator<? super T> comparator)`

Returns a stream consisting of the elements of this stream, sorted according to the provided `Comparator`.

For ordered streams, the sort is stable. For unordered streams, no stability guarantees are made.

This is a stateful intermediate operation.

Parameters:
`comparator` - a non-interfering, stateless `Comparator` to be used to compare stream elements
Returns:
the new stream

- **peek**

`Stream<T> peek(Consumer<? super T> action)`

Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

This is an intermediate operation.

For parallel stream pipelines, the action may be called at whatever time and in whatever thread the element is made available by the upstream operation. If the action modifies shared state, it is responsible for providing the required synchronization.

API Note:
This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline:

```
Stream.of("one", "two", "three", "four")
    .filter(e -> e.length() > 3)
    .peek(e -> System.out.println("Filtered value: " + e))
    .map(String::toUpperCase)
    .peek(e -> System.out.println("Mapped value: " + e))
    .collect(Collectors.toList());
```

In cases where the stream implementation is able to optimize away the production of some or all the elements (such as with short-circuiting operations like `findFirst`, or in the example described in `count()`), the action will not be invoked for those elements.

Parameters:
`action` - a non-interfering action to perform on the elements as they are consumed from the stream
Returns:

the new stream

- **limit**

`Stream<T> limit(long maxSize)`

Returns a stream consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.

This is a short-circuiting stateful intermediate operation.

API Note:

While `limit()` is generally a cheap operation on sequential stream pipelines, it can be quite expensive on ordered parallel pipelines, especially for large values of `maxSize`, since `limit(n)` is constrained to return not just any n elements, but the first n elements in the encounter order. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significant speedups of `limit()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `limit()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.

Parameters:

`maxSize` -스트림이 제한되어야하는 요소의 수

보고:

새로운 스트림

던졌습니다 :

`IllegalArgumentException`- `maxSize`음수 인 경우

- **건너 뛰기**

`스트림 < T > 건너 뛰기 (long n)`

스트림의 첫 번째 `n`요소를 제거한 후이 스트림의 나머지 요소로 구성된 스트림을 리턴 합니다. 이 스트림이보다 적은 수의 `n`요소를 포함하면 빈 스트림이 반환됩니다.

이것은 상태 저장 중간 작업 입니다.

API 참고 :

`skip()`는 일반적으로 순차 스트림 파이프 라인에서 저렴한 작업 이지만 순서가 지정된 병렬 파이프 라인에서는 비용이 많이들 수 있습니다. 특히 는 n 개 요소뿐 아니라 만남 순서에서 처음 n 개 요소 를 건너 뛰도록 제한 `n`되기 때문에 특히 . 순서가 지정되지 않은 스트림 소스 (예 :)를 사용하거나를 사용하여 순서 제약 조건을 제거하면 상황의 의미가 허용하는 경우 병렬 파이프 라인의 속도가 크게 향상 될 수 있습니다 . 발생 순서와의 일관성이 필요 하고 병렬 파이프 라인에서 성능이나 메모리 사용률이 저하 되는 경우 순차 실행으로 전환 하면 성능이 향상 될 수 있습니다.`skip(n)``generate(Supplier)``BaseStream.unordered()``skip()``skip()``BaseStream.sequential()`

매개 변수 :

`n` -건너 뛸 선행 요소의 수

보고:

새로운 스트림

던졌습니다 :

`IllegalArgumentException`- `n`음수 인 경우

- **takeWhile**

`기본 Stream < T > takeWhile ( Predicate <? super T > predicate)`

이 스트림이 정렬 된 경우 지정된 술어와 일치하는이 스트림에서 가져온 요소의 가장 긴 접두사로 구성된 스트림을 리턴합니다. 그렇지 않으면이 스트림이 순서가 지정되지 않은 경우 지정된 술어와 일치하는이 스트림에서 가져온 요소의 서브 세트로 구성된 스트림을 리턴합니다.

이 스트림이 정렬 된 경우 가장 긴 접두사는 주어진 술어와 일치하는이 스트림 요소의 연속적인 시퀀스입니다. 시퀀스의 첫 번째 요소는이 스트림의 첫 번째 요소이며 시퀀스의 마지막 요소 바로 뒤에 오는 요소는 주어진 술어와 일치하지 않습니다.

이 스트림이 순서가없고이 스트림의 일부 (전부는 아님) 요소가 주어진 술어와 일치하는 경우,이 조작의 동작은 비 결정적입니다. 빈 집합을 포함하여 일치하는 요소의 하위 집합을 자유롭게 사용할 수 있습니다.

이 스트림의 모든 요소가 주어진 술어와 일치하는 경우이 스트림의 순서가 지정되었는지 또는 순서가 지정되지 않았는 지에 관계없이이 작업은 모든 요소를 취하거나 (결과는 입력과 동일 함) 스트림의 요소가 주어진 술어와 일치하지 않으면 요소가 없습니다. 가져옵니다 (결과는 빈 스트림입니다).

이것은 단락 상태 저장 중간 작업 입니다.

API 참고 :

`takeWhile()`일반적으로 순차 스트림 파이프 라인에서는 저렴한 작업 이지만 순서가 지정된 병렬 파이프 라인에서는 작업이 유효한 접두사뿐 아니라 발생 순서에서 가장 긴 요소 접두사를 반환하도록 제한되기 때문에 비용이 많이들 수 있습니다. 순서가 지정되지 않은 스트림 소스 (예 :)를 사용하거나를 사용하여 순서 `generate(Supplier)`제약 조건을 제거하면 상황의 의미가 허용하는 경우 병렬 파이프 라인의 `BaseStream.unordered()`속도가 크게 향상 될 수 있습니다 `takeWhile()`. 발생 순서와의 일관성이 필요 `takeWhile()`하고 병렬 파이프 라인에서 성능이나 메모리 사용률이 저하 되는 경우 순차 실행으로 전환 `BaseStream.sequential()`하면 성능이 향상 될 수 있습니다.

구현 요구 사항 :

기본 구현은 `spliterator` 이 스트림의을 가져 와서 순회시이 작업의 의미를 지원하도록 해당 분할자를 래핑하고 래핑 된 분할 자와 연결된 새 스트림을 반환합니다. 반환 된 스트림은이 스트림의 실행 특성 (즉, per `BaseStream.isParallel()`) 의 실행 특성을 유지 하지만 래핑 된 분할자는 분할을 지원하지 않도록 선택할 수 있습니다. 반환 된 스트림이 닫히면 반환 된 스트림과이 스트림 모두에 대한 닫기 핸들러가 호출됩니다.

매개 변수 :

`predicate`- 요소의 가장 긴 접두사를 결정하기 위해 요소에 적용 하는 비 간섭 , 상태 비 저장 술어.

보고:

새로운 스트림

이후:

9

- **dropWhile**

```
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

Returns, if this stream is ordered, a stream consisting of the remaining elements of this stream after dropping the longest prefix of elements that match the given predicate. Otherwise returns, if this stream is unordered, a stream consisting of the remaining elements of this stream after dropping a subset of elements that match the given predicate.

If this stream is ordered then the longest prefix is a contiguous sequence of elements of this stream that match the given predicate. The first element of the sequence is the first element of this stream, and the element immediately following the last element of the sequence does not match the given predicate.

If this stream is unordered, and some (but not all) elements of this stream match the given predicate, then the behavior of this operation is nondeterministic; it is free to drop any subset of matching elements (which includes the empty set).

Independent of whether this stream is ordered or unordered if all elements of this stream match the given predicate then this operation drops all elements (the result is an empty stream), or if no elements of the stream match the given predicate then no elements are dropped (the result is the same as the input).

This is a stateful intermediate operation.

API Note:

While `dropWhile()` is generally a cheap operation on sequential stream pipelines, it can be quite expensive on ordered parallel pipelines, since the operation is constrained to return not just any valid prefix, but the longest prefix of elements in the encounter order. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significant speedups of `dropWhile()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `dropWhile()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.

Implementation Requirements:

The default implementation obtains the `spliterator` of this stream, wraps that spliterator so as to support the semantics of this operation on traversal, and returns a new stream associated with the wrapped spliterator. The returned stream preserves the execution characteristics of this stream (namely parallel or sequential execution as per `BaseStream.isParallel()`) but the wrapped spliterator may choose to not support splitting. When the returned stream is closed, the close handlers for both the returned and this stream are invoked.

Parameters:

`predicate` - a non-interfering, stateless predicate to apply to elements to determine the longest prefix of elements.

Returns:

the new stream

Since:

9

- **forEach**

```
void forEach(Consumer<? super T> action)
```

Performs an action for each element of this stream.

This is a terminal operation.

The behavior of this operation is explicitly nondeterministic. For parallel stream pipelines, this operation does not guarantee to respect the encounter order of the stream, as doing so would sacrifice the benefit of parallelism. For any given element, the action may be performed at whatever time and in whatever thread the library chooses. If the action accesses shared state, it is responsible for providing the required synchronization.

Parameters:

`action` - a non-interfering action to perform on the elements

- **forEachOrdered**

```
void forEachOrdered(Consumer<? super T> action)
```

Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.

This is a terminal operation.

This operation processes the elements one at a time, in encounter order if one exists. Performing the action for one element happens-before performing the action for subsequent elements, but for any given element, the action may be performed in whatever thread the library chooses.

Parameters:

action - a non-interfering action to perform on the elements

See Also:

forEach(Consumer)

- **toArray**

```
Object[] toArray()
```

이 스트림의 요소를 포함하는 배열을 리턴합니다.

이것은 터미널 작업 입니다.

보고:

이 스트림의 요소를 포함하는 배열

- **toArray**

```
<A> A [] toArray ( IntFunction <A []> 생성기)
```

제공된 generator함수를 사용하여 반환 된 배열을 할당하고 분할 된 실행이나 크기 조정에 필요할 수있는 추가 배열을 사용하여이 스트림의 요소를 포함하는 배열을 반환합니다 .

이것은 터미널 작업 입니다.

API 참고 :

생성기 함수는 원하는 배열의 크기 인 정수를 사용하여 원하는 크기의 배열을 생성합니다. 이것은 배열 생성자 참조로 간결하게 표현할 수 있습니다.

```
Person[] men = people.stream()
                     .filter(p -> p.getGender() == MALE)
                     .toArray(Person[]::new);
```

유형 매개 변수 :

A -결과 배열의 요소 유형

매개 변수 :

generator -원하는 유형과 제공된 길이의 새 배열을 생성하는 함수

보고:

이 스트림의 요소를 포함하는 배열

던졌습니다 :

ArrayStoreException -배열 생성기에서 반환 된 배열의 런타임 유형이이 스트림에있는 모든 요소의 런타임 유형의 상위 유형이 아닌 경우

- **줄이다**

```
T 감소 ( T identity,
         BinaryOperator < T > accumulator)
```

행하는 환원 제공된 ID 값을 사용하고,이 스트림의 요소에 연관 누적 함수 반환 감소 값. 이것은 다음과 동일합니다.

```
T result = identity;
for (T element : this stream)
    result = accumulator.apply(result, element)
return result;
```

그러나 순차적으로 실행하도록 제한되지 않습니다.

identity값은 누적 함수에 대한 식별해야한다. 이것은 모두 t에 대해 accumulator.apply(identity, t) 같음을 의미합니다 t. accumulator함수는해야 연관 기능.

이것은 터미널 작업 입니다.

API 참고 :

합계, 최소, 최대, 평균 및 문자열 연결은 모두 특수한 축소 사례입니다. 숫자의 흐름을 더하면 다음과 같이 표현할 수 있습니다.

```
Integer sum = integers.reduce(0, (a, b) -> a+b);
```

또는:

```
Integer sum = integers.reduce(0, Integer::sum);
```

이것은 단순히 루프에서 누계를 변경하는 것보다 집계를 수행하는 더 원형적인 방법으로 보일 수 있지만, 감소 작업은 추가 동기화가 필요하지 않고 데이터 경합의 위험을 크게 줄이면서 더 우아하게 병렬화됩니다.

매개 변수 :

`identity` -누적 함수의 식별 값

`accumulator` - an associative, non-interfering, stateless function for combining two values

Returns:

the result of the reduction

- **reduce**

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an `Optional` describing the reduced value, if any. This is equivalent to:

```
boolean foundAny = false;
T result = null;
for (T element : this stream) {
    if (!foundAny) {
        foundAny = true;
        result = element;
    }
    else
        result = accumulator.apply(result, element);
}
return foundAny ? Optional.of(result) : Optional.empty();
```

but is not constrained to execute sequentially.

The `accumulator` function must be an associative function.

This is a terminal operation.

Parameters:

`accumulator` - an associative, non-interfering, stateless function for combining two values

Returns:

an `Optional` describing the result of the reduction

Throws:

`NullPointerException` - if the result of the reduction is null

See Also:

`reduce(Object, BinaryOperator)`, `min(Comparator)`, `max(Comparator)`

- **reduce**

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. This is equivalent to:

```
U result = identity;
for (T element : this stream)
    result = accumulator.apply(result, element)
return result;
```

but is not constrained to execute sequentially.

The `identity` value must be an identity for the combiner function. This means that for all `u`, `combiner(identity, u)` is equal to `u`. Additionally, the `combiner` function must be compatible with the `accumulator` function; for all `u` and `t`, the following must hold:

```
combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)
```

This is a terminal operation.

API Note:

Many reductions using this form can be represented more simply by an explicit combination of `map` and `reduce` operations. The `accumulator` function acts as a fused mapper and accumulator, which can sometimes be more efficient than separate mapping and reduction, such as when knowing the previously reduced value allows you to avoid some computation.

Type Parameters:

`U` - The type of the result

Parameters:

`identity` - the identity value for the combiner function

`accumulator` - an associative, non-interfering, stateless function for incorporating an additional element into a result

`combiner` - an associative, non-interfering, stateless function for combining two values, which must be compatible with the accumulator function

Returns:

the result of the reduction

See Also:

`reduce(BinaryOperator)`, `reduce(Object, BinaryOperator)`

- **collect**

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

Performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an `ArrayList`, and elements are incorporated by updating the state of the result rather than by replacing the result. This produces a result equivalent to:

```
R result = supplier.get();
for (T element : this stream)
    accumulator.accept(result, element);
return result;
```

Like `reduce(Object, BinaryOperator)`, `collect` operations can be parallelized without requiring additional synchronization.

This is a terminal operation.

API Note:

JDK에는 시그니처가 메서드 참조와 함께 사용하기에 적합한 많은 기존 클래스가 있습니다 `collect()`. 예를 들어 다음은 문자열을에 누적합니다 `ArrayList`.

```
List<String> asList = stringStream.collect(ArrayList::new, ArrayList::add,
                                           ArrayList::addAll);
```

다음은 문자열 스트림을 가져 와서 단일 문자열로 연결합니다.

```
String concat = stringStream.collect(StringBuilder::new, StringBuilder::append,
                                     StringBuilder::append)
                            .toString();
```

유형 매개 변수 :
R -변경 가능한 결과 컨테이너의 유형
매개 변수 :
supplier-새로운 변경 가능한 결과 컨테이너를 생성하는 함수. 병렬 실행의 경우이 함수는 여러 번 호출 될 수 있으며 매번 새로운 값을 반환해야합니다.
accumulator- 요소를 결과 컨테이너로 접어야 하는 연관성 , 비 간섭 , 상태 비 저장 함수.
combiner- 두 개의 부분 결과 컨테이너를 받아들이고 병합 하는 연관성 , 비 간섭 , 상태 비 저장 함수로, 누산기 함수와 호환되어야합니다. 결합기 함수는 두 번째 결과 컨테이너의 요소를 첫 번째 결과 컨테이너로 접어야합니다.
보고:
감소의 결과

- **수집**

```
<R, A> R collect ( Collector <? super T , A, R> collector)
```

를 사용하여이 스트림의 요소에 대해 가변 감소 작업을 수행합니다 Collector. A Collector 는에 인수로 사용되는 함수를 캡슐화하여 collect(Supplier, BiConsumer, BiConsumer)수집 전략을 재사용하고 다중 수준 그룹화 또는 분할과 같은 수집 작업을 구성 할 수 있습니다.

스트림이 병렬이고 Collector is concurrent이고 스트림이 순서가 지정되지 않았거나 컬렉터가 unordered인 경우 동시 감소가 수행됩니다 (Collector동시 감소에 대한 자세한 내용 은 참조).

이것은 터미널 작업 입니다.

병렬로 실행될 때, 가변 데이터 구조의 격리를 유지하기 위해 여러 중간 결과가 인스턴스화되고, 채워지고, 병합 될 수 있습니다. 따라서 스레드로부터 안전하지 않은 데이터 구조 (예 :)와 병렬로 실행되는 경우에도 ArrayList병렬 감소를 위해 추가 동기화가 필요하지 않습니다.

API 참고 :
다음은 문자열을 ArrayList에 누적합니다.

```
List<String> asList = stringStream.collect(Collectors.toList());
```

다음은 Person도시별로 개체를 분류 합니다.

```
Map<String, List<Person>> peopleByCity
    = personStream.collect(Collectors.groupingBy(Person::getCity));
```

다음은 Person주 및 도시별로 객체를 분류 하고 두 개의 Collectors를 함께 계단식으로 분류 합니다 .

```
Map<String, Map<String, List<Person>>> peopleByStateAndCity
    = personStream.collect(Collectors.groupingBy(Person::getState,
                                                 Collectors.groupingBy(Person::getCity)));
```

유형 매개 변수 :
R -결과의 유형
A -중간 축적 유형 Collector

매개 변수 :
`collector` - `Collector`감소에 대한 설명
보고:
감소의 결과
또한보십시오:
`collect(Supplier, BiConsumer, BiConsumer)`, `Collectors`

- **분**

`Optional<T> min(Comparator<? super T> comparator)`

Returns the minimum element of this stream according to the provided `Comparator`. This is a special case of a reduction.

This is a terminal operation.

Parameters:
`comparator` - a non-interfering, stateless `Comparator` to compare elements of this stream
Returns:
an `Optional` describing the minimum element of this stream, or an empty `Optional` if the stream is empty
Throws:
`NullPointerException` - if the minimum element is null

- **max**

`Optional<T> max(Comparator<? super T> comparator)`

Returns the maximum element of this stream according to the provided `Comparator`. This is a special case of a reduction.

This is a terminal operation.

Parameters:
`comparator` - a non-interfering, stateless `Comparator` to compare elements of this stream
Returns:
an `Optional` describing the maximum element of this stream, or an empty `Optional` if the stream is empty
Throws:
`NullPointerException` - if the maximum element is null

- **count**

`long count()`

Returns the count of elements in this stream. This is a special case of a reduction and is equivalent to:

```
return mapToLong(e -> 1L).sum();
```

This is a terminal operation.

API Note:
An implementation may choose to not execute the stream pipeline (either sequentially or in parallel) if it is capable of computing the count directly from the stream source. In such cases no source elements will be traversed and no intermediate operations will be evaluated. Behavioral parameters with side-effects, which are strongly discouraged except for harmless cases such as debugging, may be affected. For example, consider the following stream:

```
List<String> l = Arrays.asList("A", "B", "C", "D");
long count = l.stream().peek(System.out::println).count();
```

The number of elements covered by the stream source, a `List`, is known and the intermediate operation, `peek`, does not inject into or remove elements from the stream (as may be the case for `flatMap` or `filter` operations). Thus the count is the size of the `List` and there is no need to execute the pipeline and, as a side-effect, print out the list elements.

Returns:

the count of elements in this stream

- **anyMatch**

`boolean anyMatch(Predicate<? super T> predicate)`

Returns whether any elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty then `false` is returned and the predicate is not evaluated.

이것은 단락 터미널 작동 입니다.

API 참고 :

이 메서드 는 스트림의 요소 (일부 x P (x)에 대해)에 대한 술어 의 실존 적 정량화 를 평가합니다 .

매개 변수 :

`predicate`- 이 스트림의 요소에 적용하기 위한 비 간섭 , 상태 비 저장 술어

보고:

`true` 스트림의 요소가 제공된 술어와 일치하는 경우, 그렇지 않으면 `false`

- **allMatch**

`boolean allMatch ( Predicate <? super T > predicate)`

이 스트림의 모든 요소가 제공된 술어와 일치하는지 여부를 리턴합니다. 결과 결정에 필요하지 않은 경우 모든 요소에 대한 술어를 평가하지 않을 수 있습니다. 스트림이 비어 있으면 `true`반환되고 술어는 평가되지 않습니다.

이것은 단락 터미널 작동 입니다.

API 참고 :

이 방법 은 스트림의 요소 (모든 x P (x)에 대해)에 대한 술어 의 보편적 정량화 를 평가합니다 . 스트림이 비어있는 경우 정량화는 막연하게 충족 되었다고하며 항상 `true`(P (x)에 관계없이)됩니다.

매개 변수 :

`predicate`- 이 스트림의 요소에 적용하기 위한 비 간섭 , 상태 비 저장 술어

보고:

`true` 스트림의 모든 요소가 제공된 술어와 일치하거나 스트림이 비어있는 경우, 그렇지 않으면 `false`

- **noneMatch**

`boolean noneMatch ( Predicate <? super T > predicate)`

이 스트림의 요소가 제공된 술어와 일치하지 않는지 여부를 리턴합니다. 결과 결정에 필요하지 않은 경우 모든 요소에 대한 술어를 평가하지 않을 수 있습니다. 스트림이 비어 있으면 `true`반환되고 술어는 평가되지 않습니다.

이것은 단락 터미널 작동 입니다.

API 참고 :

이 메서드 는 스트림의 요소에 대한 부정 술어 의 보편적 인 정량화 를 평가합니다 (모든 x ~ P (x)). 스트림이 비어 있으면 정량화가 완전히 충족되었다고하며 trueP (x)에 관계없이 항상 입니다.

매개 변수 :

predicate- 이 스트림의 요소에 적용하기 위한 비 간섭 , 상태 비 저장 술어

보고:

true 제공된 술어와 일치하는 스트림 요소가 없거나 스트림이 비어있는 경우, 그렇지 않으면 false

- **findFirst**

선택 사항 < T > findFirst ()

Optional이 스트림의 첫 번째 요소를 설명하거나 Optional스트림이 비어 있는 경우 비어 있는 것을 리턴합니다 . 스트림에 발생 순서가 없으면 모든 요소가 반환 될 수 있습니다.

이것은 단락 터미널 작동 입니다.

보고:

는 Optional빈 스트림의 첫 번째 요소를 설명하는, 또는 Optional스트림이 비어 있으면

던졌습니다 :

NullPointerException -선택된 요소가 null 인 경우

- **findAny**

선택적 < T > findAny ()

Optional스트림의 일부 요소를 설명하거나 Optional스트림이 비어 있는 경우 비어 있는 것을 리턴합니다 .

이것은 단락 터미널 작동 입니다.

The behavior of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (If a stable result is desired, use findFirst() instead.)

Returns:

an Optional describing some element of this stream, or an empty Optional if the stream is empty

Throws:

NullPointerException - if the element selected is null

See Also:

findFirst()

- **builder**

static <T> Stream.Builder<T> builder()

Returns a builder for a Stream.

Type Parameters:

T - type of elements

Returns:

a stream builder

- **empty**

static <T> Stream<T> empty()

Returns an empty sequential Stream.

Type Parameters:
`T` - the type of stream elements
Returns:
빈 순차 스트림

- **의**

`정적 <T> 스트림 <T> of (T t)`

`Stream`단일 요소를 포함 하는 순차 를 반환합니다 .

유형 매개 변수 :
`T` -스트림 요소의 유형
매개 변수 :
`t` -단일 요소
보고:
싱글 톤 순차 스트림

- **ofNullable**

`정적 <T> 스트림 <T> ofNullable (T t)`

`Stream`null이 아닌 경우 단일 요소를 포함 하는 순차 를 반환하고 , 그렇지 않으면 빈을 반환합니다 `Stream`.

유형 매개 변수 :
`T` -스트림 요소의 유형
매개 변수 :
`t` -단일 요소
보고:
지정된 요소가 널이 아닌 경우 단일 요소가있는 스트림, 그렇지 않으면 빈 스트림
이후:
9

- **의**

```
@SafeVarargs
정적 <T> 스트림 <T> of (T ... 값)
```

요소가 지정된 값인 순차적으로 정렬 된 스트림을 반환합니다.

유형 매개 변수 :
`T` -스트림 요소의 유형
매개 변수 :
`values` -새 스트림의 요소
보고:
새로운 스트림

- **반복하다**

```
정적 <T> 스트림 <T> 반복 (T 시드,
                          UnaryOperator <T> f)
```

무한 시퀀스 주문시 돌려 `Stream`함수의 반복적 적용에 의해 생성 된 `f`초기 요소 `seed`생산 `Stream`이 루어진 `seed`, `f(seed)`, `f(f(seed))`등

의 첫 번째 요소 (위치 `0`) `Stream`가 제공됩니다 `seed`. 의 경우 `n > 0`, 위치의 요소는 위치의 요소에 `n` 함수 `f`를 적용한 결과가 `n - 1`됩니다.

도포 작업 `f`한 요소가 일어나도 전에 도포하는 작업 `f`이후의 요소. 주어진 요소에 대해 라이브러리가 선택한 스레드에서 작업을 수행 할 수 있습니다.

유형 매개 변수 :
`T` -스트림 요소의 유형
매개 변수 :
`seed` -초기 요소
`f` -새로운 요소를 생성하기 위해 이전 요소에 적용 할 기능
보고:
새로운 연속 `Stream`

- **반복하다**

```
정적 <T>  스트림 <T> 반복 (T seed,
                          Predicate <? super T> hasNext,
                          UnaryOperator <T> next)
```

Returns a sequential ordered `Stream` produced by iterative application of the given `next` function to an initial element, conditioned on satisfying the given `hasNext` predicate. The stream terminates as soon as the `hasNext` predicate returns false.

`Stream.iterate` should produce the same sequence of elements as produced by the corresponding for-loop:

```
for (T index=seed; hasNext.test(index); index = next.apply(index)) {
    ...
}
```

The resulting sequence may be empty if the `hasNext` predicate does not hold on the seed value. Otherwise the first element will be the supplied `seed` value, the next element (if present) will be the result of applying the `next` function to the `seed` value, and so on iteratively until the `hasNext` predicate indicates that the stream should terminate.

도포의 액션 `hasNext`요소에 술어가 발생-전에 , 도포의 작용 `next`해당 요소로 기능한다. `next`한 요소에 대한 함수를 적용하는 작업은 후속 요소에 대한 조건자를 적용하는 작업 전에 발생`hasNext` 합니다. 주어진 요소에 대해 라이브러리가 선택한 스레드에서 작업을 수행 할 수 있습니다.

유형 매개 변수 :
`T` -스트림 요소의 유형
매개 변수 :
`seed` -초기 요소
`hasNext` -스트림이 종료되어야하는시기를 결정하기 위해 요소에 적용 할 술어.
`next` -새로운 요소를 생성하기 위해 이전 요소에 적용 할 기능
보고:
새로운 연속 `Stream`
이후:
9

- **일으키다**

```
정적 <T>  스트림 <T> 생성 ( 공급 업체 <?는 T>를 확장)
```

제공된에 의해 각 요소가 생성되는 무한 순차 비 순차 스트림을 반환합니다 `Supplier`. 이것은 상수 스트림, 임의 요소의 스트림 등을 생성하는 데 적합합니다.

유형 매개 변수 :
`T` -스트림 요소의 유형
매개 변수 :
`s` - `Supplier`생성 된 요소의
보고:
순서없는 새로운 무한 순차 `Stream`

- **연결**

```
static <T>  Stream <T> concat ( Stream <? extends T> a,
                                Stream <? extends T> b)
```

첫 번째 스트림의 모든 요소와 두 번째 스트림의 모든 요소가 뒤 따르는 요소가있는 느리게 연결된 스트림을 만듭니다. 결과 스트림은 두 입력 스트림이 모두 정렬 된 경우 정렬되고 입력 스트림 중 하나가 병렬 인 경우 병렬로 정렬됩니다. 결과 스트림이 닫히면 두 입력 스트림 모두에 대한 닫기 핸들러가 호출됩니다.

구현 참고 사항 :

반복 된 연결에서 스트림을 구성 할 때주의하십시오. 깊게 연결된 스트림의 요소에 액세스하면 깊은 콜 체인 또는 `StackOverflowError`.

반환 된 스트림의 순차 / 병렬 실행 모드에 대한 후속 변경 사항은 입력 스트림으로 전파된다는 보장이 없습니다.

유형 매개 변수 :

`T` -스트림 요소의 유형

매개 변수 :

`a` -첫 번째 스트림

`b` -두 번째 스트림

보고:

두 입력 스트림의 연결