(http://baeldung.com)

Introducción a Spliterator en Java

Última modificación: 30 de enero de 2018

por Michal Aibin (http://www.baeldung.com/author/michal-author/) (http://www.baeldung.com/author/michal-author/)

Java (http://www.baeldung.com/category/java/) +

1. Información general

La interfaz del *Spliterator* , introducida en Java 8, se puede **usar para secuencias de desplazamiento y particionamiento** . Es una utilidad básica para *Streams* , especialmente para los paralelos.

En este artículo, cubriremos su uso, características, métodos y cómo crear nuestras propias implementaciones personalizadas.

2. *Spliterator* API

2.1. *tryAdvance*

Este es el método principal utilizado para recorrer una secuencia. El método **toma un *Consumidor* que se utiliza para consumir elementos del *Spliterator* uno por uno secuencialmente** y devuelve *falso* si no hay elementos para atravesar.

Aquí, veremos cómo usarlo para atravesar y particionar elementos.

Primero, supongamos que tenemos un *ArrayList* con 35000 artículos y que la clase de *artículo* se define como:

```
public class Article {
    private List<Author> listOfAuthors;
    private int id;
    private String name;

    // standard constructors/getters/setters
}
```

Ahora, implementemos una tarea que procese la lista de artículos y agregue un sufijo de " - *publicado por Baeldung"* al nombre de cada artículo:

```
public String call() {
    int current = 0;
    while (spliterator.tryAdvance(a -> a.setName(article.getName()
      .concat("- published by Baeldung")))) {
        current++;
    }

    return Thread.currentThread().getName() + ":" + current;
}
```

Tenga en cuenta que esta tarea genera la cantidad de artículos procesados cuando finaliza la ejecución.

Otro punto clave es que utilizamos el método *tryAdvance ()* para procesar el siguiente elemento.

2.2. *trySplit*

A continuación, dividamos *Spliterators* (de ahí el nombre) y procesamos las particiones de forma independiente.

El método *trySplit* intenta dividirlo en dos partes. Luego, los elementos del proceso llamador y, finalmente, la instancia devuelta procesarán los otros, permitiendo que los dos se procesen en paralelo.

Generemos nuestra lista primero:

```
public static List<Article> generateElements() {
    return Stream.generate(() -> new Article("Java"))
      .limit(35000)
      .collect(Collectors.toList());
}
```

A continuación, obtenemos nuestra instancia de *Spliterator* utilizando el método *spliterator ()* . Luego aplicamos nuestro método *trySplit ()* :

```
@Test
public void givenSpliterator_whenAppliedToAListOfArticle_thenSplittedInHalf() {
    Spliterator<Article> split1 = Executor.generateElements().spliterator();
    Spliterator<Article> split2 = split1.trySplit();

    assertThat(new Task(split1).call())
      .containsSequence(Executor.generateElements().size() / 2 + "");
    assertThat(new Task(split2).call())
      .containsSequence(Executor.generateElements().size() / 2 + "");
}
```

El proceso de división funcionó según lo previsto y dividió los registros por igual .

2.3. *estimadoTamaño*

El método *estimadoSize* nos da una cantidad estimada de elementos:

```
LOG.info("Size: " + split1.estimateSize());
```

Esto dará como resultado:

```
Size: 17500
```

2.4. *tieneCaracterísticas*

Esta API verifica si las características dadas coinciden con las propiedades del *Spliterator.* Entonces, si invocamos el método anterior, la salida será una representación *int* de esas características:

```
LOG.info("Characteristics: " + split1.characteristics());
```

```
Characteristics: 16464
```

3. Características del *Spliterator*

Tiene ocho características diferentes que describen su comportamiento. Esos pueden usarse como sugerencias para herramientas externas:

- ***TAMAÑO*** - si es capaz de devolver una cantidad exacta de elementos con elmétodo *estimateSize ()*
- ***CLASIFICADO*** : si se itera a través de una fuente clasificada
- ***SUBSIZED*** - Si dividimos la instancia usando un *trySplit ()* método y obtener Spliterators que son *SIZED* así
- ***CONCURRENTE*** : si la fuente se puede modificar de forma segura al mismo tiempo
- ***DISTINCT*** - si para cada par de elementos encontrados *x, y,! X.equals (y)*
- ***INMUTABLE*** : si los elementos que ***posee la*** fuente no se pueden modificar estructuralmente
- ***NONNULL*** - si la fuente contiene nulos o no
- ***ORDERED*** - si itera sobre una secuencia ordenada

4. Un *Spliterator* personalizado

4.1. Cuándo personalizar

Primero, supongamos el siguiente escenario:

Tenemos una clase de artículo con una lista de autores y el artículo que puede tener más de un autor. Además, consideramos a un autor relacionado con el artículo si la identificación de su artículo relacionado concuerda con la identificación del artículo.

Nuestra clase de *Autor* se verá así:

```
public class Author {
    private String name;
    private int relatedArticleId;

    // standard getters, setters & constructors
}
```

A continuación, implementaremos una clase para contar autores al atravesar una secuencia de autores. Luego, **la clase realizará una reducción** en la transmisión.

Echemos un vistazo a la implementación de la clase:

```
public class RelatedAuthorCounter {
    private int counter;
    private boolean isRelated;

    // standard constructors/getters

    public RelatedAuthorCounter accumulate(Author author) {
        if (author.getRelatedArticleId() == 0) {
            return isRelated ? this : new RelatedAuthorCounter( counter, true);
        } else {
            return isRelated ? new RelatedAuthorCounter(counter + 1, false) : this;
        }
    }

    public RelatedAuthorCounter combine(RelatedAuthorCounter RelatedAuthorCounter) {
        return new RelatedAuthorCounter(
          counter + RelatedAuthorCounter.counter,
          RelatedAuthorCounter.isRelated);
    }
}
```

Cada método en la clase anterior realiza una operación específica para contar al atravesar.

First, the ***accumulate()*** **method traverse the authors one by one in an iterative way**, then ***combine()*** **sums two counters using their values**. Finally, the *getCounter()* returns the counter.

Now, to test what we've done so far. Let's convert our article's list of authors to a stream of authors:

```
Stream<Author> stream = article.getListOfAuthors().stream();
```

And implement a ***countAuthor()*** **method to perform the reduction on the stream using** ***RelatedAuthorCounter***.

```
private int countAutors(Stream<Author> stream) {
    RelatedAuthorCounter wordCounter = stream.reduce(
      new RelatedAuthorCounter(0, true),
      RelatedAuthorCounter::accumulate,
      RelatedAuthorCounter::combine);
    return wordCounter.getCounter();
}
```

If we used a sequential stream the output will be as expected *"count = 9"*, however, the problem arises when we try to parallelize the operation.

Let's take a look at the following test case:

```
@Test
void
  givenAStreamOfAuthors_whenProcessedInParallel_countProducesWrongOutput() {
    assertThat(Executor.countAutors(stream.parallel())).isGreaterThan(9);
}
```

Aparentemente, algo salió mal: dividir la secuencia en una posición aleatoria provocó que un autor se contara dos veces.

4.2. Cómo personalizar

Para resolver esto, debemos **implementar un *Spliterator* que divida a los autores solo cuando coincidan *id* y *articleId***. Aquí está la implementación de nuestro *Spliterator* personalizado :

```
public class RelatedAuthorSpliterator implements Spliterator<Author> {
    private final List<Author> list;
    AtomicInteger current = new AtomicInteger();
    // standard constructor/getters

    @Override
    public boolean tryAdvance(Consumer<? super Author> action) {
        action.accept(list.get(current.getAndIncrement()));
        return current.get() < list.size();
    }

    @Override
    public Spliterator<Author> trySplit() {
        int currentSize = list.size() - current.get();
        if (currentSize < 10) {
            return null;
        }
        for (int splitPos = currentSize / 2 + current.intValue();
          splitPos < list.size(); splitPos++) {
            if (list.get(splitPos).getRelatedArticleId() == 0) {
                Spliterator<Author> spliterator
                  = new RelatedAuthorSpliterator(
                  list.subList(current.get(), splitPos));
                current.set(splitPos);
                return spliterator;
            }
        }
        return null;
    }

    @Override
    public long estimateSize() {
        return list.size() - current.get();
    }

    @Override
    public int characteristics() {
        return CONCURRENT;
    }
}
```

Ahora aplicar el método *countAuthors ()* dará el resultado correcto. El siguiente código demuestra que:

```
@Test
public void
  givenAStreamOfAuthors_whenProcessedInParallel_countProducesRightOutput() {
    Stream<Author> stream2 = StreamSupport.stream(spliterator, true);

    assertThat(Executor.countAutors(stream2.parallel())).isEqualTo(9);
}
```

Además, el *Spliterator* personalizado se crea a partir de una lista de autores y atraviesa el mismo manteniendo la posición actual.

Discutamos con más detalles la implementación de cada método:

- ***tryAdvance*** - pasa los autores al *consumidor* en la posición actual del índice e incrementa su posición
- ***trySplit*** *:* define el mecanismo de división, en nuestro caso, el *RelatedAuthorSpliterator* se crea cuando los ids coinciden, y la división divide la lista en dos partes
- ***estimatedSize*** - es la diferencia entre el tamaño de la lista y la posición del autor actualmente iterado
- ***características*** - devuelve lascaracterísticasdel *Spliterator* , en nuestro caso *SIZED* ya que el valor devuelto por elmétodo *estimadoSize ()* es exacto; además, *CONCURRENT* indica que la fuente de este *Spliterator* puede ser modificada con seguridad por otros hilos

5. Soporte para valores primitivos

La ***API Spliterator*** **admite valores primitivos que incluyen** ***double*** **,** ***int*** **y** ***long*** .

La única diferencia entre utilizar un *Spliterator* dedicado genérico y uno primitivo es el *Consumidor* dado y el tipo de *Spliterator* .

Por ejemplo, cuando lo necesitamos para un valor *int* , necesitamos pasar un *intConsumer* . Además, aquí hay una lista de *Spliterators* dedicados primitivos :

- *OfPrimitive <T, T_CONS, T_SPLITR extiende Spliterator.OfPrimitive <T, T_CONS, T_SPLITR >>* : interfaz principal para otras primitivas
- *OfInt* : Un *Spliterator* especializado para *int*
- *OfDouble* : Un *Spliterator* dedicado para el *doble*
- *OfLong* : Un *Spliterator* dedicado por *mucho tiempo*

6. Conclusión

En este artículo, ***cubrimos el*** **uso de** Java 8 ***Spliterator*** **, métodos, características, proceso de división, soporte primitivo y cómo personalizarlo.**

Como siempre, la implementación completa de este artículo se puede encontrar en Github (https://github.com/eugenp/tutorials/tree/master/core-java-8) .

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

Suscribir ▾ ▲ el más nuevo ▲ más antiguo ▲ el más votado

Huésped

Rafal

No hay un código fuente usado en este artículo en github. Además, 'trySplit' no está muy bien descrito, no se invoca en ejemplos de código.

+ 0 −

Hace 2 meses

(http://www.baeldung.com/author/loredana-crusoveanu/)

Editor

Loredana Crusoveanu (http://www.baeldung.com/author/loredana-crusoveanu/)

Hola Rafal,

Hemos actualizado el ejemplo del código para que sea más claro. Además, puede encontrar el código fuente en el paquete spliteratorAPI.

Aclamaciones.

+ 0 −

Hace 2 meses

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)