W

Tutorial de Java 8 Stream

31 de julio de 2014

Este tutorial basado en ejemplos proporciona una visión general detallada de las secuencias de Java 8. Cuando leí por primera vez acerca de la `Stream` API, que estaba confundido acerca del nombre ya que suena similar a `InputStream` y `OutputStream` desde Java I / O. Pero las transmisiones de Java 8 son algo completamente diferente. Las transmisiones son Mónadas , por lo que juegan un papel importante en la *programación funcional* de Java:

> *En la programación funcional, una mónada es una estructura que representa cálculos definidos como secuencias de pasos. Un tipo con una estructura de mónada define lo que significa encadenar operaciones o anidar funciones de ese tipo.*

Esta guía le enseña cómo trabajar con flujos de Java 8 y cómo utilizar los diferentes tipos de operaciones de flujo disponibles. Aprenderá sobre el orden de procesamiento y cómo el orden de las operaciones de transmisión afecta el rendimiento del tiempo de ejecución. Las operaciones de flujo más potentes `reduce` , `collect` y `flatMap` se cubren en detalle. El tutorial termina con una mirada en profundidad a las corrientes paralelas.

Si aún no está familiarizado con las expresiones lambda de Java 8, las interfaces funcionales y las referencias de métodos, es probable que desee leer primero mi Tutorial de Java 8 antes de comenzar con este tutorial.

Cómo funcionan las corrientes

Un flujo representa una secuencia de elementos y admite diferentes tipos de operaciones para realizar cálculos sobre esos elementos:

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

// C1
// C2
```

Las operaciones de flujo son intermedias o terminales. Las operaciones intermedias devuelven un flujo para que podamos encadenar múltiples operaciones intermedias sin utilizar puntos y coma. Las operaciones de la terminal son nulas o devuelven un resultado no continuo. En el ejemplo anterior `filter` , `map` y `sorted` son operaciones intermedias mientras que `forEach` es una operación terminal. Para obtener una lista completa de todas las operaciones de flujo disponibles, consulte el Javadoc de flujo . Tal cadena de operaciones de flujo como se ve en el ejemplo anterior también se conoce como *canalización de operación* .

La mayoría de las operaciones de flujo aceptan algún tipo de parámetro de expresión lambda, una interfaz funcional que especifica el comportamiento exacto de la operación. La mayoría de esas operaciones deben ser *sin interferencia* y *sin estado* . Qué significa eso?

Una función no interfiere cuando no modifica el origen de datos subyacente del flujo, por ejemplo, en el ejemplo anterior, ninguna expresión lambda se modifica `myList` agregando o eliminando elementos de la colección.

Una función no tiene estado cuando la ejecución de la operación es determinista, por ejemplo, en el ejemplo anterior, ninguna expresión lambda depende de cualquier variable o estado mutable del ámbito externo que pueda cambiar durante la ejecución.

Diferentes tipos de arroyos

Las secuencias se pueden crear desde varias fuentes de datos, especialmente colecciones. Las listas y los conjuntos admiten nuevos métodos `stream()` y `parallelStream()` para crear una secuencia o una secuencia paralela. Las transmisiones paralelas son capaces de operar en múltiples subprocesos y

```
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

Llamar al método `stream()` en una lista de objetos devuelve una secuencia de objetos normal. Pero no tenemos que crear colecciones para trabajar con flujos como vemos en el siguiente ejemplo de código:

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

Solo utiliza `Stream.of()` para crear una secuencia de un montón de referencias de objetos.

Además de las secuencias de objetos regulares, Java 8 se entrega con tipos especiales de secuencias para trabajar con los tipos de datos primitivos `int`, `long` y `double`. Como habrás adivinado que es `IntStream`, `LongStream` y `DoubleStream`.

IntStreams puede reemplazar el bucle for común utilizando `IntStream.range()`:

```
IntStream.range(1, 4)
    .forEach(System.out::println);

// 1
// 2
// 3
```

Todas esas corrientes primitivas funcionan igual que las corrientes de objetos regulares con las siguientes diferencias: Las corrientes primitivas utilizan expresiones lambda especializadas, por ejemplo, en `IntFunction` lugar de `Function` o en `IntPredicate` lugar de `Predicate`. Y los flujos primitivos soportan las operaciones agregadas de terminales adicionales `sum()` y `average()`:

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println);  // 5.0
```

A veces es útil transformar un flujo de objetos regular en un flujo primitivo o viceversa. Para ese propósito, las secuencias de objetos admiten las operaciones de mapeo especiales `mapToInt()`, `mapToLong()` y `mapToDouble`:

```
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

Los flujos primitivos se pueden transformar en flujos de objetos a través de `mapToObj()`:

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

Este es un ejemplo combinado: la secuencia de dobles se asigna primero a una secuencia int y luego se asigna a una secuencia de objetos de cadenas:

```
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

Orden de procesamiento

Ahora que hemos aprendido cómo crear y trabajar con diferentes tipos de flujos, profundicemos en cómo se procesan las operaciones de flujos bajo el capó.

Una característica importante de las operaciones intermedias es la pereza. Mira esta muestra donde falta una operación de terminal:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

Al ejecutar este fragmento de código, no se imprime nada en la consola. Esto se debe a que las

Extendamos el ejemplo anterior mediante la operación del terminal `forEach`:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

La ejecución de este fragmento de código da como resultado el resultado deseado en la consola:

```
filter:  d2
forEach: d2
filter:  a2
forEach: a2
filter:  b1
forEach: b1
filter:  b3
forEach: b3
filter:  c
forEach: c
```

El orden del resultado puede ser sorprendente. Un enfoque ingenuo sería ejecutar las operaciones horizontalmente una tras otra en todos los elementos del flujo. Pero en cambio cada elemento se mueve verticalmente a lo largo de la cadena. La primera cadena "d2" pasa `filter` entonces `forEach`, solo entonces se procesa la segunda cadena "a2".

Este comportamiento puede reducir el número real de operaciones realizadas en cada elemento, como vemos en el siguiente ejemplo:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });

// map:      d2
// anyMatch: D2
// map:      a2
// anyMatch: A2
```

La operación `anyMatch` retorna `true` tan pronto como el predicado se aplica al elemento de entrada

cadena de flujo, `map` solo debe ejecutarse dos veces en este caso. Así que en lugar de mapear todos los elementos de la corriente, `map` se llamará lo menos posible.

Por qué el orden importa

El siguiente ejemplo se compone de dos operaciones intermedias `map` y `filter` y el funcionamiento del terminal `forEach`. Vamos a inspeccionar una vez más cómo se están ejecutando esas operaciones:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));

// map:     d2
// filter:  D2
// map:     a2
// filter:  A2
// forEach: A2
// map:     b1
// filter:  B1
// map:     b3
// filter:  B3
// map:     c
// filter:  C
```

Como puede haber adivinado ambos `map` y `filter` se le llama cinco veces por cada cadena en la colección subyacente, mientras `forEach` que solo se llama una vez.

Podemos reducir considerablemente el número real de ejecuciones si cambiamos el orden de las operaciones, moviéndonos `filter` al principio de la cadena:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
```

```
// filter:  d2
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

Ahora, `map` solo se llama una vez, por lo que el flujo de la operación se realiza mucho más rápido para un mayor número de elementos de entrada. Tenlo en cuenta al componer complejas cadenas de métodos.

Ampliemos el ejemplo anterior mediante una operación adicional, `sorted`:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

La clasificación es un tipo especial de operación intermedia. Se trata de una *operación* llamada con *estado* ya que para ordenar una colección de elementos debe mantener el estado durante el pedido.

La ejecución de este ejemplo da como resultado el siguiente resultado de la consola:

```
sort:    a2; d2
sort:    b1; a2
sort:    b1; d2
sort:    b1; a2
sort:    b3; b1
sort:    b3; d2
sort:    c; b3
sort:    c; d2
filter:  a2
map:     a2
forEach: A2
filter:  b1
filter:  b3
```

Primero, la operación de clasificación se ejecuta en toda la colección de entrada. En otras palabras `sorted` se ejecuta horizontalmente. En este caso, `sorted` se llama ocho veces para combinaciones múltiples en cada elemento de la colección de entrada.

Una vez más podemos optimizar el rendimiento al reordenar la cadena:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// filter:  b1
// filter:  b3
// filter:  c
// map:     a2
// forEach: A2
```

En este ejemplo, `sorted` nunca se ha llamado porque `filter` reduce la colección de entrada a solo un elemento. Por lo tanto, el rendimiento aumenta considerablemente para las colecciones de entrada más grandes.

Reutilizando Streams

Las secuencias de Java 8 no se pueden reutilizar. Tan pronto como llame a cualquier operación de terminal, el flujo se cerrará:

```
Stream<String> stream =
    Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

Llamar `noneMatch` después `anyMatch` en el mismo flujo da como resultado la siguiente excepción:

```
java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
    at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
    at com.winterbe.java8.Streams5.test7(Streams5.java:38)
    at com.winterbe.java8.Streams5.main(Streams5.java:28)
```

Para superar esta limitación, debemos crear una nueva cadena de flujos para cada operación de terminal que queremos ejecutar, por ejemplo, podríamos crear un proveedor de flujos para construir un nuevo flujo con todas las operaciones intermedias ya configuradas:

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Cada llamada a `get()` construir un nuevo flujo en el que estamos guardados para llamar a la operación de terminal deseada.

Operaciones avanzadas

Las transmisiones soportan muchas operaciones diferentes. Ya hemos aprendido sobre las operaciones más importantes como `filter` o `map`. Lo dejo a usted para que descubra todas las demás operaciones disponibles (vea Stream Javadoc). En su lugar, vamos a profundizar en las operaciones más complejas `collect`, `flatMap` y `reduce`.

La mayoría de los ejemplos de código de esta sección utilizan la siguiente lista de personas para fines de demostración:

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
```

```
}

List<Person> persons =
    Arrays.asList(
        new Person("Max", 18),
        new Person("Peter", 23),
        new Person("Pamela", 23),
        new Person("David", 12));
```

Recoger

Collect es una operación de terminal extremadamente útil para transformar los elementos del flujo en un tipo diferente de resultado, por ejemplo `List`, a , `Set` o `Map`. Collect acepta un `Collector` que consta de cuatro operaciones diferentes: un *proveedor* , un *acumulador* , un *combinador* y un *finalizador* . Esto suena muy complicado al principio, pero la mejor parte es que Java 8 admite varios colectores integrados a través de la `Collectors` clase. Por lo tanto, para las operaciones más comunes, no es necesario que usted mismo implemente un recopilador.

Vamos a empezar con un caso de uso muy común:

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);    // [Peter, Pamela]
```

Como puede ver, es muy sencillo construir una lista a partir de los elementos de un flujo. Necesita un conjunto en lugar de una lista, solo use `Collectors.toSet()`.

El siguiente ejemplo agrupa a todas las personas por edad:

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);     // 19.0
```

Si está interesado en estadísticas más completas, los recopiladores de resumen devuelven un objeto de estadísticas de resumen incorporado especial. Así que simplemente podemos determinar *min* , *max* y la aritmética *media de* edad de las personas, así como la *suma* y *contar* .

```
IntSummaryStatistics ageSummary =
    persons
        .stream()
        .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

El siguiente ejemplo une a todas las personas en una sola cadena:

```
String phrase = persons
    .stream()
    .filter(p -> p.age >= 18)
    .map(p -> p.name)
    .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

El recopilador de unión acepta un delimitador, así como un prefijo y sufijo opcionales.

Para transformar los elementos de flujo en un mapa, debemos especificar cómo se deben asignar las claves y los valores. Tenga en cuenta que las claves asignadas deben ser únicas, de lo contrario `IllegalStateException` se lanzará una. Opcionalmente, puede pasar una función de combinación como un parámetro adicional para omitir la excepción:

```
Map<Integer, String> map = persons
    .stream()
    .collect(Collectors.toMap(
        p -> p.age,
        p -> p.name,
        (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

Ahora que conocemos a algunos de los coleccionistas incorporados más potentes, intentemos construir nuestro propio colector especial. Queremos transformar a todas las personas del flujo en una sola cadena que consta de todos los nombres en letras superiores separadas por el `|` carácter de canalización. Para conseguirlo creamos un nuevo coleccionista `Collector.of()`. Tenemos que pasar los cuatro ingredientes de un colector: un *proveedor* , un *acumulador* , un *combinador* y un *finalizador* .

```
Collector<Person, StringJoiner, String> personNameCollector =
    Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
    .stream()
    .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID
```

Dado que las cadenas en Java son inmutables, necesitamos una clase auxiliar `StringJoiner` que permita al colector construir nuestra cadena. El proveedor inicialmente construye un StringJoiner con el delimitador apropiado. El acumulador se utiliza para agregar el nombre en mayúsculas de cada persona al StringJoiner. El combinador sabe cómo combinar dos StringJoiners en uno. En el último paso, el finalizador construye la cadena deseada desde el StringJoiner.

FlatMap

Ya hemos aprendido cómo transformar los objetos de un flujo en otro tipo de objetos utilizando la `map` operación. El mapa es un poco limitado porque cada objeto solo puede asignarse a otro objeto exactamente. Pero, ¿qué pasa si queremos transformar un objeto en varios otros o ninguno en absoluto? Aquí es donde `flatMap` viene al rescate.

FlatMap transforma cada elemento del flujo en un flujo de otros objetos. Por lo tanto, cada objeto se transformará en cero, uno o varios objetos respaldados por secuencias. El contenido de esos flujos se colocará en el flujo devuelto de la `flatMap` operación.

Antes de que veamos `flatMap` en acción necesitamos una jerarquía de tipos apropiada:

```
class Foo {
    String name;
    List<Bar> bars = new ArrayList<>();
```

```
        this.name = name;
    }
}

class Bar {
    String name;

    Bar(String name) {
        this.name = name;
    }
}
```

A continuación, utilizamos nuestro conocimiento sobre las secuencias para instanciar un par de objetos:

```
List<Foo> foos = new ArrayList<>();

// create foos
IntStream
    .range(1, 4)
    .forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars
foos.forEach(f ->
    IntStream
        .range(1, 4)
        .forEach(i -> f.bars.add(new Bar("Bar" + i + " <- " + f.name))));
```

Ahora tenemos una lista de tres foos cada uno que consta de tres barras.

FlatMap acepta una función que debe devolver un flujo de objetos. Así que para resolver los objetos de barra de cada foo, simplemente pasamos la función apropiada:

```
foos.stream()
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));

// Bar1 <- Foo1
// Bar2 <- Foo1
// Bar3 <- Foo1
// Bar1 <- Foo2
// Bar2 <- Foo2
// Bar3 <- Foo2
// Bar1 <- Foo3
// Bar2 <- Foo3
// Bar3 <- Foo3
```

Como puede ver, hemos transformado con éxito el flujo de tres objetos foo en un flujo de objetos de

Finalmente, el ejemplo de código anterior se puede simplificar en una sola línea de operaciones de flujo:

```
IntStream.range(1, 4)
    .mapToObj(i -> new Foo("Foo" + i))
    .peek(f -> IntStream.range(1, 4)
        .mapToObj(i -> new Bar("Bar" + i + " <- " f.name))
        .forEach(f.bars::add))
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));
```

FlatMap también está disponible para la `Optional` clase introducida en Java 8. La `flatMap` operación Optionals devuelve un objeto opcional de otro tipo. Por lo tanto, puede ser utilizado para prevenir `null` cheques desagradables .

Piense en una estructura altamente jerárquica como esta:

```
class Outer {
    Nested nested;
}

class Nested {
    Inner inner;
}

class Inner {
    String foo;
}
```

Para resolver la cadena interna `foo` de una instancia externa, debe agregar varias comprobaciones nulas para evitar posibles NullPointerExceptions:

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner != null) {
    System.out.println(outer.nested.inner.foo);
}
```

Se puede obtener el mismo comportamiento utilizando `flatMap` operaciones opcionales :

```
Optional.of(new Outer())
    .flatMap(o -> Optional.ofNullable(o.nested))
    .flatMap(n -> Optional.ofNullable(n.inner))
    .flatMap(i -> Optional.ofNullable(i.foo))
    .ifPresent(System.out::println);
```

Cada llamada a `flatMap` devuelve un `Optional` envoltorio del objeto deseado si está presente o `null` si está ausente.

Reducir

La operación de reducción combina todos los elementos del flujo en un solo resultado. Java 8 soporta tres tipos diferentes de `reduce` métodos. El primero reduce un flujo de elementos a exactamente un elemento del flujo. Veamos cómo podemos usar este método para determinar la persona de mayor edad:

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);    // Pamela
```

El `reduce` método acepta una `BinaryOperator` función de acumulador. Eso es en realidad un lugar `BiFunction` donde ambos operandos comparten el mismo tipo, en ese caso `Person`. Las funciones son similares `Function` pero aceptan dos argumentos. La función de ejemplo compara las edades de las dos personas para devolver a la persona con la edad máxima.

El segundo `reduce` método acepta tanto un valor de identidad como un `BinaryOperator` acumulador. Este método se puede utilizar para construir una nueva Persona con los nombres agregados y las edades de todas las demás personas en el flujo:

```
Person result =
    persons
        .stream()
        .reduce(new Person("", 0), (p1, p2) -> {
            p1.age += p2.age;
            p1.name += p2.name;
            return p1;
        });

System.out.format("name=%s; age=%s", result.name, result.age);
// name=MaxPeterPamelaDavid; age=76
```

El tercer `reduce` método acepta tres parámetros: un valor de identidad, un `BiFunction` acumulador y una función combinadora de tipo `BinaryOperator`. Dado que el tipo de valores de identidad no está restringido al `Person` tipo, podemos utilizar esta reducción para determinar la suma de edades de todas las personas:

```
        .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);

    System.out.println(ageSum);  // 76
```

Como puede ver, el resultado es *76* , pero ¿qué sucede exactamente debajo del capó? Extendamos el código anterior por alguna salida de depuración:

```
    Integer ageSum = persons
        .stream()
        .reduce(0,
            (sum, p) -> {
                System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
                return sum += p.age;
            },
            (sum1, sum2) -> {
                System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
                return sum1 + sum2;
            });

    // accumulator: sum=0; person=Max
    // accumulator: sum=18; person=Peter
    // accumulator: sum=41; person=Pamela
    // accumulator: sum=64; person=David
```

Como se puede ver la función del acumulador hace todo el trabajo. Primero se llama con el valor de identidad inicial *0* y la primera persona *Máx* . En los siguientes tres pasos `sum` aumenta continuamente según la edad de los últimos pasos, hasta una edad total de *76* .

¿Espera qué? El combinador nunca se llama? Ejecutar el mismo flujo en paralelo levantará el secreto:

```
    Integer ageSum = persons
        .parallelStream()
        .reduce(0,
            (sum, p) -> {
                System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
                return sum += p.age;
            },
            (sum1, sum2) -> {
                System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
                return sum1 + sum2;
            });

    // accumulator: sum=0; person=Pamela
    // accumulator: sum=0; person=David
    // accumulator: sum=0; person=Max
    // accumulator: sum=0; person=Peter
    // combiner: sum1=18; sum2=23
    // combiner: sum1=23; sum2=12
```

La ejecución de esta secuencia en paralelo da como resultado un comportamiento de ejecución completamente diferente. Ahora el combinador se llama en realidad. Dado que el acumulador se llama en paralelo, se necesita el combinador para resumir los valores acumulados por separado.

Vamos a profundizar en corrientes paralelas en el siguiente capítulo.

Arroyos paralelos

Las secuencias se pueden ejecutar en paralelo para aumentar el rendimiento del tiempo de ejecución en una gran cantidad de elementos de entrada. Las transmisiones paralelas utilizan un común `ForkJoinPool` disponible a través del `ForkJoinPool.commonPool()` método estático . El tamaño del grupo de subprocesos subyacentes utiliza hasta cinco subprocesos, dependiendo de la cantidad de núcleos de CPU físicos disponibles:

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());    // 3
```

En mi máquina, el pool común se inicializa con un paralelismo de 3 por defecto. Este valor se puede reducir o aumentar configurando el siguiente parámetro JVM:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Las colecciones apoyan el método `parallelStream()` para crear un flujo paralelo de elementos. Alternativamente, puede llamar al método intermedio `parallel()` en un flujo dado para convertir un flujo secuencial en una contraparte paralela.

Para subestimar el comportamiento de ejecución paralela de un flujo paralelo, el siguiente ejemplo imprime información sobre el hilo actual para `sout` :

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n"
```

Al investigar la salida de depuración, deberíamos obtener una mejor comprensión de qué subprocesos se utilizan realmente para ejecutar las operaciones de flujo:

```
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:     c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:     b1 [main]
forEach: B1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:     a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

Como puede ver, la transmisión paralela utiliza todos los subprocesos disponibles del común `ForkJoinPool` para ejecutar las operaciones de transmisión. La salida puede diferir en ejecuciones consecutivas porque el comportamiento de ese subproceso en particular realmente no es determinista.

Ampliemos el ejemplo de una operación de flujo adicional, `sort`:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n",
            s1, s2, Thread.currentThread().getName());
        return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

El resultado puede parecer extraño al principio:

```
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:     b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
map:     a1 [ForkJoinPool.commonPool-worker-2]
map:     c2 [ForkJoinPool.commonPool-worker-3]
sort:    A2 <> A1 [main]
sort:    B1 <> A2 [main]
sort:    C2 <> B1 [main]
sort:    C1 <> C2 [main]
sort:    C1 <> B1 [main]
sort:    C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

Parece que `sort` se ejecuta secuencialmente en el hilo principal solamente. En realidad, `sort` en un flujo paralelo utiliza el nuevo método Java 8 `Arrays.parallelSort()` bajo el capó. Como se indica en Javadoc, este método decide la longitud de la matriz si la clasificación se realizará de forma secuencial o en paralelo:

> *Si la longitud de la matriz especificada es menor que la granularidad mínima, entonces se ordena utilizando el método Arrays.sort apropiado.*

Volviendo al `reduce` ejemplo de la última sección. Ya descubrimos que la función del combinador solo se llama en paralelo pero no en flujos secuenciales. Veamos que hilos están realmente involucrados:

```
List<Person> persons = Arrays.asList(
    new Person("Max", 18),
    new Person("Peter", 23),
    new Person("Pamela", 23),
    new Person("David", 12));

persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s [%s]\n",
                sum, p, Thread.currentThread().getName());
            return sum += p.age;
```

```
            System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",
                sum1, sum2, Thread.currentThread().getName());
            return sum1 + sum2;
        });
```

La salida de la consola revela que tanto el *acumulador* como las funciones del *combinador* se ejecutan en paralelo en todos los subprocesos disponibles:

```
accumulator: sum=0; person=Pamela; [main]
accumulator: sum=0; person=Max;    [ForkJoinPool.commonPool-worker-3]
accumulator: sum=0; person=David;  [ForkJoinPool.commonPool-worker-2]
accumulator: sum=0; person=Peter;  [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=18; sum2=23;     [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=23; sum2=12;     [ForkJoinPool.commonPool-worker-2]
combiner:    sum1=41; sum2=35;     [ForkJoinPool.commonPool-worker-2]
```

En resumen, se puede afirmar que las transmisiones paralelas pueden brindar un buen aumento de rendimiento a las transmisiones con una gran cantidad de elementos de entrada. Pero tenga en cuenta que algunas operaciones de flujo paralelo como `reduce` y `collect` necesitan cálculos adicionales (operaciones combinadas) que no son necesarias cuando se ejecutan secuencialmente.

Además, hemos aprendido que todas las operaciones de transmisión paralela comparten el mismo común de JVM `ForkJoinPool`. Por lo tanto, es probable que desee evitar la implementación de operaciones de secuencia de bloqueo lento, ya que esto podría ralentizar otras partes de su aplicación que dependen en gran medida de las secuencias paralelas.

Eso es todo

Mi guía de programación para las secuencias de Java 8 termina aquí. Si está interesado en obtener más información sobre las secuencias de Java 8, le recomiendo la documentación del paquete Stream Javadoc . Si desea obtener más información sobre los mecanismos subyacentes, es probable que desee leer el artículo de Martin Fowlers sobre los Oleoductos de recolección .

Si también está interesado en JavaScript, puede echar un vistazo a Stream.js , una implementación de JavaScript de la API de Java 8 Streams. También puedes leer mi Tutorial de Java 8 y mi Tutorial de Nashorn de Java 8 .

Esperemos que este tutorial te haya sido útil y que hayas disfrutado leyéndolo. El código fuente completo de los ejemplos de tutoriales está alojado en GitHub . Siéntase libre de bifurcar el repositorio o enviarme sus comentarios a través de Twitter .

Benjamin es ingeniero de software, desarrollador de Full Stack en Pondus , un corredor emocionado y jugador de futbolín. Ponte en contacto en Twitter y GitHub .

Lee mas

Reciente | Todos los mensajes | Java | Tutoriales

Tutorial Java 11

Migrar proyectos Maven a Java 11

Tutorial De Secuencia De Kotlin

Integración de React.js en aplicaciones web jQuery existentes