Spliteratoren

Sven Eric Panitz

Parallelisierte Iteration

- Streams können parallel über ihren Bereich iterieren.
- Wie machen die das?
- Wie wird die Arbeit verteilt?
- Wer macht die eigentliche Arbeit?
- Was lässt sich alles parallelisieren?

Manche Arbeit kann gut geteilt werden

Beispiel:
10 Leute bekommen den Auftrag einen Graben von 100 Meter Länge, ein Meter Breite und 1 Meter Tiefe auszuheben.

Manche Arbeit kann nicht gut geteilt werden

Beispiel:
10 Leute bekommen den Auftrag einen Graben von 1 Meter Länge, ein Meter Breite und 100 Meter Tiefe auszuheben.

Parallel Iterieren

- Oft ist die Reihenfolge, mit der durch bestimmte Elemente iteriert wird, gleichgültig.
- Zum Beispiel:
  Durch eine Liste von Kundenadressen wird iteriert, um jedem eine Angebotsmail zu verschicken.
  Die Reihenfolge, in der hier iteriert wird, in der also die Emails verschickt werden, spielt keine Rolle.
- Solche Iterationen können gefahrlos nebeneinander her geschehen.

Von Iterator zum Spliterator

- Seit Java 1.8 gibt es eine neue Schnittstelle für Iterationen: Spliterator
- Diese kann nicht nur Iterieren sondern bietet auch die Möglichkeit an, auf Wunsch die Iteration auf einen weiteren Spliterator zu verteilen, also die Arbeit auf zwei Objekte zu verteilen.

Iteration im Spliterator

- Statt next() und hasNext() in der Schnittstelle Iterator hat die Schnittstelle Spliterator nur eine Methode:
- `boolean tryAdvance(Consumer<? super E> cons);`
- Der Rückgabetyp gibt an, ob es noch ein weiteres Element zum Abarbeiten gab. (also quasi die Aufgabe von hasNext())
  Der Parameter gibt an, was mit jedem Element gemacht werden soll.

Iterieren mit Iterator vs Spliterator

```
for (Iterator<String> it=someIterator; it.hasNext();){
  String x = it.next();
  do something smart with x;
}
```

```
Spliterator<String> spl = someSpliterator;
while (spl.tryAdvance
                    (x → do something smart with x));
```

Ein abstrakter Spliterator

Zusätzlich zur Methode `tryAdvance` mit der die Iteration beschrieben wird gibt es drei weitere abstrakte Methoden.

Sie können zunächst einmal wie folgt implementiert werden.

```
public abstract class AbstractSpliterator<T> implements Spliterator<T> {
  @Override public Spliterator<T> trySplit() {return null;}
  @Override public long estimateSize() {return Long.MAX_VALUE;}
  @Override public int characteristics() {return 0;}
}
```

Beispiel: ein Spliterator für einen Array

Als Beispiel sei eine Klasse umgesetzt, die als Spliterator über die Elemente eines Arrays iteriert.

```
public class FirstSplit<A> extends AbstractSpliterator<A>{
  A[] xs;
  int start, end;

  public FirstSplit(A[] xs) {
    this(xs,0,xs.length);
  }
  public FirstSplit(A[] xs,int start, int end) {
    this.start = start;
    this.end = end;
    this.xs = xs;
  }
```

Beispiel: tryAdvance für einen Array

tryAdvance, testet ob es noch ein nächstes Element gibt, wendet dann den Consumer auf dieses an und schaltet einen Schritt weiter:

```
@Override
public boolean tryAdvance(Consumer<? super A> action){
  if (start>=end) return false;
  action.accept(xs[start]);
  start=start+1;
  return true;
}
```

Direkte Iteration mit Spliterator

```
var as = new String[]
         {"das","Pferd","frisst","keinen","Gurkensalat"};
var xs = new FirstSplit<>(as);

while
 (xs.tryAdvance(x→System.out.println(x.toUpperCase())));
```

Spliterator als Grundlage für einen Stream

Ein Spliteratorobjekt kann als Grundlage für einen Stream verwendet werden.
Hierzu gibt es in der Klasse `StreamSupport` die Methode `stream`.

```
Integer[] bs = new Integer[]
   {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
var is = new FirstSplit<>(bs);
var s = StreamSupport.stream(is, false);
s.forEach(x->System.out.println(x));
```

Der Spiterator bewerkstelligt die Arbeit der eigentlichen Iteration.

Spliterator, der sich splitten kann

Will man eine Parallelisierung ermöglichen, muss man in der Lage sein, die Arbeit auf zwei Spliteratoren zu verteilen.
Durch Implementierung der Methode trySplit kann diese Aufteilung definiert werden.

```
@Override public Spliterator<A> trySplit() {
  var n = end-start;
  if (n<5) return null;
  var middle = start+n/2;
  var result = new FirstSplit<>(xs, middle,end);
  end = middle;
  return result;
}
```

trySplit

- trySplit erzeugt einen neuen Spliterator der einen Teil der Arbeit übernimmt.
- Der ursprüngliche Spliterator ist so zu ändern, dass er nicht mehr den abgegebenen Teil der Arbeit macht.
- Lässt sich die Arbeit nicht mehr sinnvoll aufteilen, so wird null zurück gegeben und der ursprüngliche Spliterator bleibt unverändert.

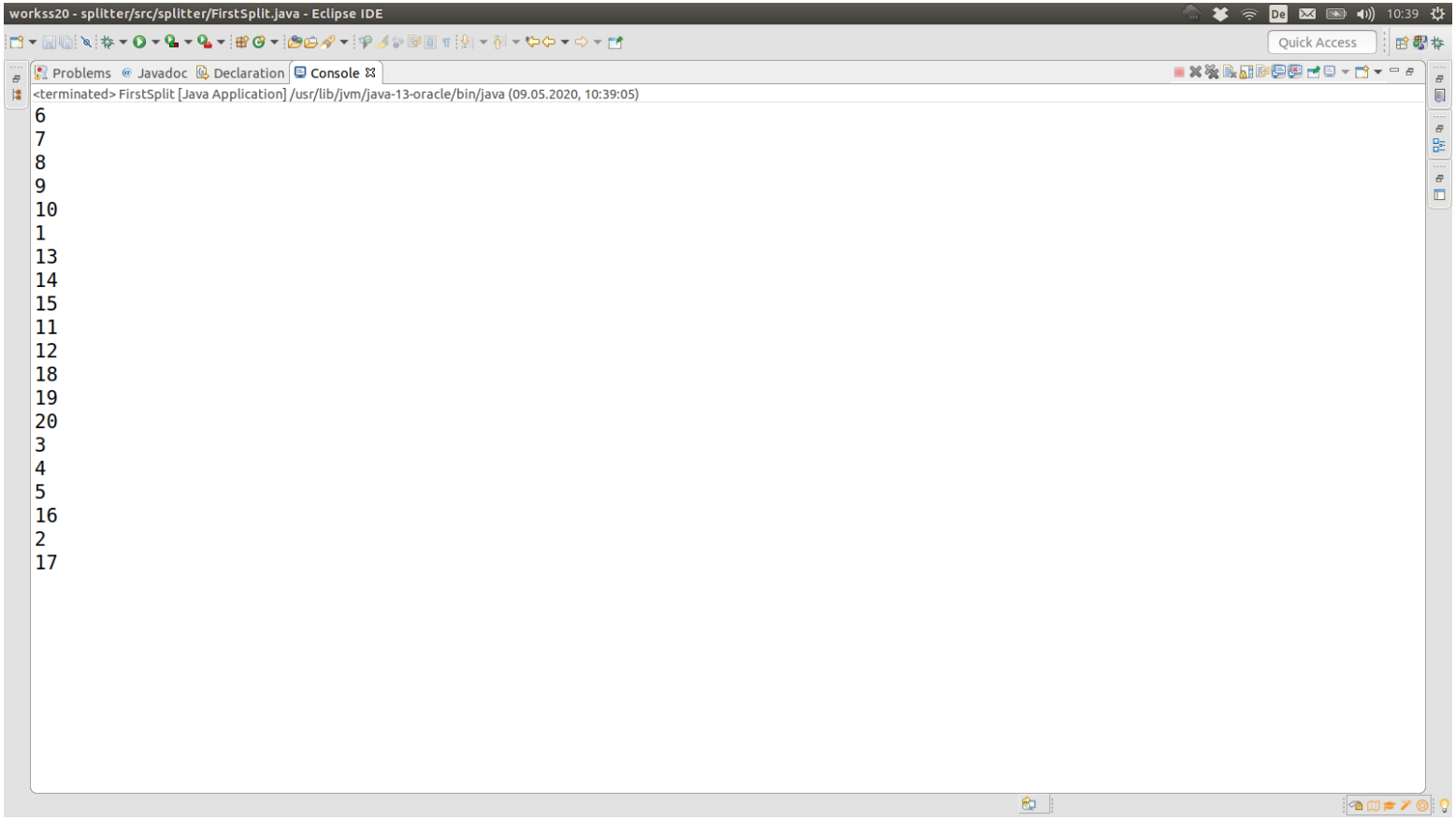
Spliterator für einen parallelen Stream

Jetzt kann man der Methode stream der Klasse StreamSupport im zweiten Parameter das bool`sche Flag mitgeben, das anzeigt, dass die Iteration parallel angegangen werden darf.

```
Integer[] bs = new Integer[]
   {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
var is = new FirstSplit<>(bs);
var s = StreamSupport.stream(is, true);
s.forEach(x->System.out.println(x));
```

Der so erzeugte Stream wird den Spliterator bitten, sich zu splitten und parallele Threads zur Iteration starten.

Beispielausgabe der parallelen Iteration



Methoden für Eigenschaften des Spliterator

- Es gibt zwei Methoden, die zusätzliche Informationen zu einem Spliterator geben.
- Diese sollen helfen, einzuschätzen, ob und wann sinnvoll parallelisiert werden kann.
- Hierzu lassen sich: `estimateSize()` und `characteristics()` implementieren.

estimateSize

- `estimateSize()` gibt eine Abschätzung über die Anzahl der zu iterierenden Objekte.
- Wenn diese nicht bekannt ist, wird `Long.MAX_VALUE` zurück gegeben.
- Beispiel in unserem Fall:

```
@Override public long estimateSize() {
  return end-start;
}
```

`int` characteristics()

- In einer Zahl sind bitweise weitere Eigenschaften codiert.
- Die Methode `int characteristics()` zeigt diese Eigenschaften an.
- Hierfür gibt es Konstanten, die die einzelnen Eigenschaften setzen:
  - static int CONCURRENT
  - static int DISTINCT
  - static int IMMUTABLE
  - static int NONNULL
  - static int ORDERED
  - static int SIZED
  - static int SORTED
  - static int SUBSIZED

static interface	Spliterator.OfPrimitive<T,T_CONS,T_SPLITR extends Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>> A Spliterator specialized for primitive values.

Field Summary

Fields

Modifier and Type	Field and Description
static int	**CONCURRENT** Characteristic value signifying that the element source may be safely concurrently modified (allowing additions, replacements, and/or removals) by multiple threads without external synchronization.
static int	**DISTINCT** Characteristic value signifying that, for each pair of encountered elements `x, y, !x.equals(y)`.
static int	**IMMUTABLE** Characteristic value signifying that the element source cannot be structurally modified; that is, elements cannot be added, replaced, or removed, so such changes cannot occur during traversal.
static int	**NONNULL** Characteristic value signifying that the source guarantees that encountered elements will not be `null`.
static int	**ORDERED** Characteristic value signifying that an encounter order is defined for elements.
static int	**SIZED** Characteristic value signifying that the value returned from `estimateSize()` prior to traversal or splitting represents a finite size that, in the absence of structural source modification, represents an exact count of the number of elements that would be encountered by a complete traversal.
static int	**SORTED** Characteristic value signifying that encounter order follows a defined sort order.
static int	**SUBSIZED** Characteristic value signifying that all Spliterators resulting from `trySplit()` will be both **SIZED** and **SUBSIZED**.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Zusammenfassung

- Spliteratoren sind die Arbeitstiere der Streams.
- Sie definieren, wie über einen Quellbereich iteriert wird.
- Sie definieren, wie die Iteration aufgeteilt werden kann.
- Sie geben weitere Eigenschaften über den Quellbereich an.